NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

FORENSIC ANALYSIS OF WINDOW'S® VIRTUAL MEMORY INCORPORATING THE SYSTEM'S PAGE-FILE

by

Jared M. Stimson

December 2008

Thesis Advisor:	Chris S. Eagle
Second Reader:	George W. Dinolt

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** December 2008	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE: Forensic Analysis of Windows® Virtual Memory Incorporating the System's Page-File.

5. FUNDING NUMBERS

6. AUTHOR(S) Jared Stimson

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES)
N/A

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE
A

13. ABSTRACT (maximum 200 words)

Computer Forensics is concerned with the use of computer investigation and analysis techniques in order to collect evidence suitable for presentation in court. The examination of volatile memory is a relatively new but important area in computer forensics. More recently criminals are becoming more forensically aware and are now able to compromise computers without accessing the hard disk of the target computer. This means that traditional incident response practice of pulling the plug will destroy the only evidence of the crime. While some techniques are available for acquiring the contents of main memory, few exist which can analyze these data in a meaningful way. One reason for this is how memory is managed by the operating system. Data belonging to one process can be distributed arbitrarily across physical memory or the hard disk, making it very difficult to recover useful information. This report will focus on how these disparate sources of information can be combined to give a single, contiguous address space for each process. Using address translation a tool is developed to reconstruct the virtual address space of a process by combining a physical memory dump with the page-file on the hard disk.

14. SUBJECT TERMS

Computer Forensics, Page-file, Virtual Memory

15. NUMBER OF PAGES
93

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

THIS PAGE INTENTIONALLY LEFT BLANK

FORENSIC ANALYSIS OF WINDOW'S® VIRTUAL MEMORY INCORPORATING THE SYSTEM'S PAGE-FILE

Jared M. Stimson
Lieutenant Junior Grade, United States Navy
B.S., University of Nebraska, 2006

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 2008

Author: Jared M. Stimson

Approved by: Chris S. Eagle
Thesis Advisor

George W. Dinolt
Second Reader

Peter J. Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Computer Forensics is concerned with the use of computer investigation and analysis techniques in order to collect evidence suitable for presentation in court. The examination of volatile memory is a relatively new but important area in computer forensics. More recently criminals are becoming more forensically aware and are now able to compromise computers without accessing the hard disk of the target computer. This means that traditional incident response practice of pulling the plug will destroy the only evidence of the crime. While some techniques are available for acquiring the contents of main memory, few exist which can analyze these data in a meaningful way. One reason for this is how memory is managed by the operating system. Data belonging to one process can be distributed arbitrarily across physical memory or the hard disk, making it very difficult to recover useful information. This report will focus on how these disparate sources of information can be combined to give a single, contiguous address space for each process. Using address translation a tool is developed to reconstruct the virtual address space of a process by combining a physical memory dump with the page-file on the hard disk.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank my advisor Professor Chris Eagle, George Dinolt, and Tim Vidas. They gave much in advice and guidance, and without their help this project would not have been possible. Professor Eagle has been an exceptional Professor and mentor**.**

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. CYBER CRIME

Security experts say that there was an alarming rise in the number of sophisticated attacks recorded in 2006. They were made possible, partly, due to a large increase in the number of vulnerabilities identified in popular software applications because last year brought a sharp increase in the number of software security vulnerabilities discovered and actively exploited by attackers. In 2006, Microsoft Corp. issued 97 security related patches that the company designated "critical", meaning attackers could use them to exploit vulnerable machines. In comparison, Microsoft issued only 37 critical updates in 2005. Also, fourteen of last year's critical flaws were "zero day" threats, meaning Microsoft first learned about the bugs only after they were already being exploited.

Another alarming development is the level of sophistication attackers are now using to develop their attacks. Some exploits are able to reside exclusively in volatile memory without ever accessing the hard drive. When a piece of software can do this it will leave no evidence on the system once it is shutdown. One exception to this is if pieces of the process are paged to the disk. However, at the time of writing the paging file structure is not fully understood. Research has been done to increase the information that the paging file can provide, but further investigation is required to understand what can be obtained using it. The harsh reality is that without evidence of its activity and origin, authorities have no way of stopping these attacks. Also, researchers fear that attacks will become even more advanced in 2008. "Criminals have gone from trying to hit as many machines as possible to focusing on techniques that allow them to remain undetected on infected machines longer," Symantec's Weafer said.

Some software security vendors suspect that a new Trojan horse program, dubbed "Rustock.B" by some anti-virus companies, may serve as the template for future malware attacks. The program modifies itself slightly each time it installs on a new machine to evade detection by anti-virus software. In addition, it hides in the deep in the Windows© operating system, creates invisible copies of itself, and refuses to run under common malware analysis tools to resist identification and analysis by security researchers. "This is about the nastiest piece of malware we've ever seen, and we're going to be seeing more of it," said Alex Eckelberry, president of Clearwater, Florida-based security vendor Sunbelt Software. "The new threats that we saw in 2006 have shown us that the malware authors are ingenious and creative in their methods. Unfortunately, those attributes aren't ones we would normally consider laudable in the context of criminals."

B. MOTIVATION

Computer Forensics is a field concerned with the use of tools and techniques to locate and collect digital evidence to be used in court. Evidence might be collected during and after a crime or misuse. [1] Such evidence may be found in both logical (files, caches, buffers, etc.) and physical places (hard disks, CDs/DVDs, removable media, main memory, etc.). While collecting evidence, Anzaldua, Godwin, and Volonino state the best practice is to unplug a computer or remove a laptop’s battery, so as to preserve the exact contents of a disk for later investigation without introducing changes to the system. Using the operating system shutdown alters log and temp file states; furthermore, a shutdown may trigger a logic bomb and a possible data wipe. [1] However, this approach has a problem; it does not preserve active system memory. A portion of the volatile memory can be found in the remnants of the operating system swap file, but this is not enough to reconstruct the state of the system. Thus, there is a push to find a way to collect a forensic image of memory without significantly altering the state of the system.

From the Order of Volatility listed in RFC 3227, the first item of volatile data that should be collected during live-response is the contents of physical memory. [1] Although the specifics of collecting particular parts of volatile memory, such as network connections or running processes, have been known for some time, the issue of collecting, parsing, and analyzing the entire contents of physical memory is a relatively new endeavor. The most important questions when looking at memory are: why collect the contents of RAM, how is it useful, and what is missed when it is not analyzed?

Some investigators collect the contents of RAM in hope of finding something not on the hard drive, such as strings or passwords, and malware analysts look to memory when dealing with malware because it is often encrypted or obfuscated in such a way that static analysis is extremely difficult. Finally, rootkits can hide processes, files, Registry keys, and even network connections from the tools usually used to enumerate them, however, analyzing the contents of RAM can reveal these hidden items. The image may also contain information about processes that have since exited, but still reside in memory. When pages of memory are used by a process and the process terminates, these pages are marked as free, but the data is not overwritten until the pages need to be reused.

As physical memory analysis grows, more investigators will pursue this as a viable source of valuable information. However, what is currently lacking is a tool which can merge the various sources that make up a virtual address space into a format that can be later analyzed in an efficient manner. [1] This report will investigate what data is stored and where it can be recovered from. In order to do this, it will examine the virtual memory system used by the Windows© XP Operating System so that a tool can be developed.

C. OVERVIEW

Inspired by the thesis by John S. Schultz at NPS titled "Offline Forensic Analysis of MICROSOFT® Windows® XP Physical Memory" [7], this thesis will expand upon the work done by Schultz to create an open source tool to analyze Windows© XP physical memory dumps for digital forensic evidence. This section begins with a short overview of the current state of cyber crime and the motivation behind this report. Chapter II opens with the current state of computer forensics and the moves on to show how to collect the contents of physical memory and the page-file. In Section III, the Windows© Virtual Memory system is explained, showing how it is possible to recover information from a Windows© XP memory image. This includes an examination of virtual to physical address translation and the paging system, and how to navigate the virtual address space. Section IV introduces the important structures resident in memory, and how they can be analyzed to find useful information. Once the contents of memory are understood, the software used to automate the analysis of the image is covered in Chapter V. Finally, a summary of the findings of this report and recommendations for further work are given in Section VI.

THIS PAGE INTENTIONALLY LEFT BLANK

II. COMPUTER FORENSICS

As attacks become more sophisticated, it is necessary to advance the latest tools and techniques used in forensic science to meet the increasing demands of investigators. In cyber crime investigations, the crime scene may consist of many computers perhaps spanning several networks. A cyber criminal may affect a system locally or remotely. Where as a local attacker may leave physical evidence, a remote attack from the Internet can come from anywhere in the world and will leave only electronic evidence. In either case, digital forensic evidence can be gathered from the criminal's computer, the victim's computer, or both. This digital evidence can be broadly grouped into two categories, volatile and non-volatile.

Non-volatile electronic evidence can be recovered after a system is powered off and is found on hard disks, USB flash drives, and removable media. It is in non-volatile media where most of the electronic evidence originates. System logs, malicious code, internet browser cache, e-mails, and deleted files are all forms of non-volatile evidence. Network logs may contain TCP session logs indicating the source IP address from where the attack originated. The malicious code may be analyzed to determine what the attacker did to the system and analysis of the hard disk can also lead to the recovery of deleted files, which may contain evidence. E-mail and browser history can show the criminal’s intent, expose any accomplices, and even give evidence as to how the attack occurred.

However, a careful cyber criminal may permanently erase any incriminating evidence from non-volatile media, making its recovery impossible. In the case of a compromised computer running malicious code, there is other evidence that can be useful; evidence in volatile memory. Unlike data stored on hard drives, evidence found in main memory contains useful information about each running process, such as create times of various system objects, exit times, open files, executing code, and child process. However, it disappears once power is removed from the system. This type of evidence is useful if a malicious program is running on a live system, because unlike the files on the hard drive, this evidence cannot be erased from memory as long as malicious code is running. If the investigator can capture the state of the system as closely to the way it was when the incident occurred, then through careful analysis evidence can be taken from the system. The ultimate goal is to be able to completely reconstruct the state of the system using the memory and hard disks. If this can be accomplished then the crime scene can be reconstructed and analyzed in a controlled environment, making gathering evidence more reliable.

A. CONVENTIONAL ANALYSIS

There are many places where digital evidence might be found within a computer system. One procedure for incident response is to immediately remove power from the system and take it to a computer forensics lab. [15] On the other hand, if the system is powered off before volatile storage such as physical memory is captured its content is lost. Once a computer system is in the lab, a write blocker is used to further prevent writing to the disks. Once that is in place, forensic copies of the system's hard disks are made and future analysis is carried out using the copies rather than the original disks. [13] A forensic copy is an exact copy which does not alter the contents of the source disk. This is done so that the results of the investigation can be reproduced and verified by independent investigators. Write blockers may be implemented using software after the system has been booted; however the safest method is a hardware device installed between the hard disk and the mother board.

A forensic investigation needs to be carried out fairly quickly because the evidence is usually needed for a court case, and if the evidence is not found in time it becomes useless. As hard disk capacities increase, investigators need to focus their efforts in order to find useful evidence. Fortunately, the contents of physical memory may be able to provide investigators important evidence or insight as to where to find it on the disk; thus reducing the amount of time and manpower needed for an investigation. As physical memory analysis expands, more investigators will pursue this as a viable source of valuable information. Unfortunately, what is currently lacking are extensive tools which can merge the various sources that make up a virtual address space in to a format that can be later analyzed in an efficient manner.

B. DUMPING PHYSICAL MEMORY

According to guidelines, digital evidence should be collected in order from the most volatile to the least. [13] This means that the one of the first things an investigator should do is copy the contents of RAM from all computers at the scene that are still running. The problem is that nothing should be done which might compromise the state of the suspect device. This reasoning is sound; however, this rule may be relaxed as long as the impact of the imaging technique is considered in the investigation. [16] There are several methods to acquire physical memory and this section provides an overview of the various options available as well as the technical aspects related to each.

1. Hardware Devices

In the last few years, there has been research into alternatives to software-based acquisition. These techniques have the advantage of gaining direct access to the memory via DMA that can copy the entire contents of RAM without modifying it. In addition, they can copy it very quickly while halting the processor, which eliminates the risk of data changing during the imaging process.

In February 2004, Brian Carrier and Joe Grand published “A Hardware-Based Memory Acquisition Procedure for Digital Investigations,” in the Digital Investigation Journal. This paper proposed the idea of using a hardware expansion card named Tribble to acquire the contents of physical memory. [2] It dumps physical memory in a manner that does not introduce new software to the system and does not rely on the operating system. This allows an investigator to retrieve the contents of volatile memory without starting a new process or running potentially malicious code. The paper introduced a prototype which can be installed in a PCI slot and copies the contents of the RAM to external media. A hardware device such as the Tribble is unobtrusive and easily accessible. The major limitation with a device such as this is that if not installed prior to an incident it is ineffective for incident response. [2] Currently, the Tribble device is not commercially available; however, other hardware devices do exist, but they are only available to security professionals.

Research preformed by Maximillian Dornseif has shown that it is possible to access memory through the Firewire interface. At PacSec in November of 2005, he presented a paper, “Owned by an iPod” where he demonstrated how a firewire device can read/write active memory within a Mac, BSD or Linux machine. At Ruxcon 2006, Adam Boileau presented “Hit by a Bus: Physical Access Attacks with Firewire” where he enabled the targeting of a Windows© XP machine. Utilizing a Linux box with firewire support he revealed several hacks using live memory reading and writing against Windows© XP. However, firewire collection presents some problems. First, there may not be firewire ports available on the computer and second, plugging in a firewire device might require the operating system to activate the port, which may alter the state of the system and if not done properly, may crash the system. The primary issue is that the concepts and tools available for firewire memory imaging are still immature and need more research before they will become widely acceptable. Active memory and live data collection are still new to the field of digital forensics and have issues regarding reliability and analysis. As recently as BlackHat in February 2007 in DC, Joanna Rutkowska demonstrated how to defeat firewire based memory imaging utilizing a low level program, in the CPU. [6] Thus, the desire to collect a snapshot of the state of the system should not overshadow the value of preserving the integrity of the data collected. [6]

2. Software Methods

When using software acquisition the imaging program must first be loaded into memory, which alters the memories contents and possibly displaces data. Also, the program could rely on libraries that may have been subverted by the attacker. It is possible that the program could receive false data from corrupted system files since the software accesses physical memory through the operating system. Thus, with software alone, it is not possible to obtain a snapshot of physical memory completely unaffected by such tools. However, an undisturbed snapshot is not always necessary for forensic analysis of memory because useful forensic evidence can be extracted from a physical memory image captured with a software tool. [3] Since there are no native tools on Windows© XP to dump the contents of system memory, a third-party program must be used. One such tool is George Garner's variant of the GNU dd utility [3]. The dd program copies data in blocks from one file to another. In the case of Garner's version, the input file is \\.\PhysicalMemory, which represents the device name for physical memory in Windows©. This, however, is no longer available on Vista, the newest version of Windows©. It is important to use the noerror option when reading from memory to ensure all readable blocks of data are copied because without this option, the dd program will stop after a read error. [3] As for the output file, it is recommended that no output file is specified in favor of piping the output of *dd* to *netcat* with the command line. The *netcat*[18] program can be used to send and receives data over a network connection, this allows physical memory to be copied and sent directly to an external computer connected to the network so the data is not saved to the local hard disk. A newer variant of *dd* is now available called *KnTDD* from GMG Systems, Inc. and is part of KnTTools which provides advanced memory acquisition and analysis capabilities[19].

3. Hibernation

Hibernation is used to save the contents of volatile memory to disk in a file called the hiberfil.sys which allows the system to power down without permanently losing that data. Since the system uses this file to restore the state of the system, some useful information might be available in the hibernation file. It is also possible that that hibernation could be a useful way to acquire the contents of memory, since no additional programs would have to be loaded. Unfortunately though, there are several problems associated with using hibernation. First, when the system is hibernated it writes data to a file, hiberfil.sys and if this file exists it will be overwritten, therefore the old file should be copied from the system before the system is hibernated. Second, enabling hibernation on systems where it is disabled requires changing the state of the machine, which increases the risk of changing data. Microsoft Windows supports a "suspend to disk" mode. Nicolas Ruff and Matthieu Suiche developed a library, called Sandman, which allows reading and writing the hibernation file. They recently presented their results at PacSec 07 [20].

Hibernation could come in handy in a forensic investigation. You just have to send the computer to sleep in order to preserve the most important parts of its main memory on disk. The presentation outlines the format of the hibernation file. It also briefly discusses the variations between different OS versions and mentions the compression algorithms involved. With the help of the Sandman library the hibernation file could be converted into a raw memory dump and possibly also into a crash dump (because the CPU state is preserved, too). Unfortunately the library is not available to the public yet.

C. ACQUIRING THE PAGE-FILE

Even more pages can be recovered if the examiner has access to the page-file for each system. Using the page-file might also allow the examiner to use more information in the memory image. In order to use the page-file effectively, however, it must be acquired at the same time as memory acquisition. By default, Windows© uses only one paging file, *%SystemDrive%\pagefile.sys*, but the true locations and filenames for paging files should be found using the registry. Capturing the page-file is difficult on a live system as traditional file copying utilities cannot open them. When the system is running the files are in use by the operating system and may not be opened by another process. It would be beneficial for first responders to have a program to capture both physical memory and the paging file in one step, but this will require further research. To acquire the page-file mount the drive in a guest operating system and copy the page-file. The examiner must be careful to use a write blocker so that no changes are made to the source drive. The delay between capturing a memory image and the paging file may create inconsistencies that could hinder analysis. Power to the system should be secured immediately after obtaining the memory image in order to freeze the page-file in a state as close to the state it was in when the image was obtained.

III. WINDOWS VIRTUAL MEMORY

The Windows© operating system creates a private virtual memory space for each running process using the memory manager. The memory manager is used to map a process' virtual address space to the system's physical memory or RAM. All modern operating systems use some form of Virtual Memory (VM) system to give each process a large address space while preventing it from gaining access to data belonging to other processes. [5] The basic concept behind virtual memory is relatively simple. For each process, the operating system maps virtual addresses to physical memory because RAM has less storage space and is more expensive than a hard disk. The memory manager creates data structures called page tables which the CPU uses to translate virtual addresses into physical addresses. Each virtual address is associated with a system-space structure called a page table entry (PTE), which contains the physical address to which the virtual address is mapped. [16] For example, Figure 3-1 shows how three consecutive virtual pages are mapped to three pages that are not physically contiguous.

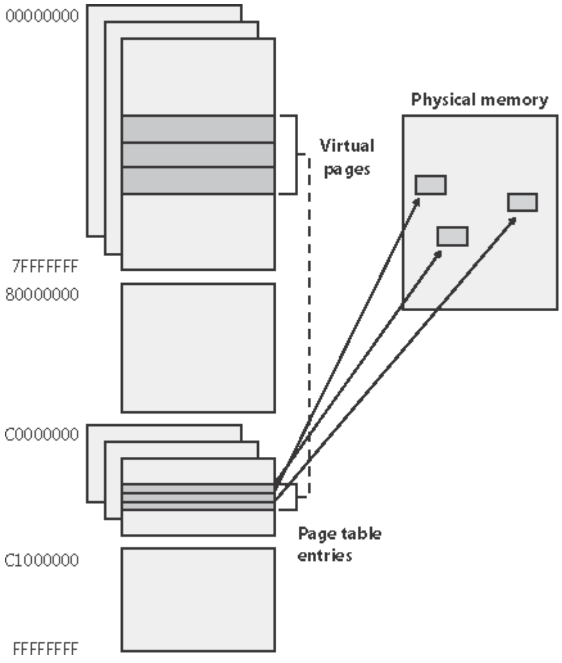

Figure 3.1 Mapping Virtual Addresses to Physical Memory (x86) [From 5]

If the operating system runs out of space in physical RAM, some data is paged out to make room. The paged data is maintained on the system's hard drive. [5] If a page is not mapped into physical memory, the operating system marks the page as invalid. Any access to this page causes a page fault, which then causes the OS to copy the contents of the page from secondary storage into memory.

To implement VM, Windows© maintains a large amount of data. It needs to know if data is in RAM or on the disk. Maintaining this information for each byte of an address space would require more memory than the address space itself and for this reason Windows© breaks the address space into 4KB pages (or 4MB if large pages are enabled) and maintains this information in page tables. A page table entry (PTE) consists of the physical address of the page if the page is mapped to RAM and also some attributes of the page. [15] The VMM used by Windows© XP allows each process to access a full 4GB of virtual addresses, translating those virtual addresses into physical addresses. [5] Within each virtual address space, there is a portion dedicated as user space and a portion reserved for the operating system. User space is for application code, global variables, the process stack, and dynamically linked libraries (DLLs) and it spans from virtual address 0x00000000 to 0x7FFFFFFF. The system address space is accessible by all processes and is for use by the operating system. It ranges from 0x80000000 to 0xFFFFFFFF and contains the necessary information for system management of the virtual memory, including the page directory and the page table entries (PTEs) used for virtual addresses translation.

A. VIRTUAL ADDRESS TRANSLATION

Windows© uses virtual addresses to abstract the memory storage system from the rest of the operating system and other programs. The operating system presents each program with a large private virtual address space and each time a program references a virtual address the operating system translates that virtual address into a physical address and retrieves the requested data. If the data is not in memory, it loads the data from the disk. During memory analysis, the examiner needs to use this same translation process, but without help from the operating system. Windows© on an x86 system uses a two-level page table structure to translate virtual addresses to physical addresses. [5] A 32-bit virtual address is interpreted as three separate components—the page directory index, the page table index, and the byte index—that are used as offsets into the structures that describe page how the page is mapped, as illustrated in Figure 3-2. The page size and the PTE width dictate the width of the page directory and page table index fields. For example, on x86 systems, the byte index requires 12 bits to describe the location of all 4096 bytes in each page.

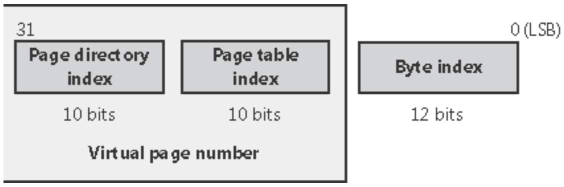

Figure 3.2 Translating a Valid Virtual Address (x86-specific) [From 5]

The page directory index is used to locate the page table in which the PTE is located. The page table index is used to locate the PTE, which, contains the physical address to which a virtual page maps. The byte index indicates the proper address within that physical page. Figure 3.3 shows the relationship of these three values and how they are used to map a virtual address into a physical address.

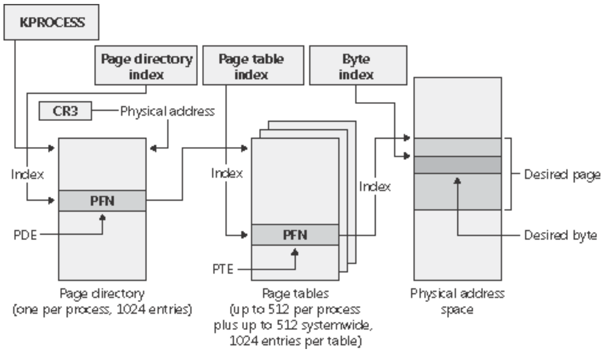

Figure 3.3 Components of a 32-bit Virtual Address on x86 Systems [From 5]

Address translation is generally a three stage procedure. Every process on a Windows system maintains a DirectoryTableBase variable. On an x86 systems this value is stored in the CR3 register when the process is running. This value contains the base address of the table of Page Directory Entries (PDE) for that process. For each virtual address, a PDE is specified using a few bits from the original virtual address. The PDE is used to find the base address of a page of Page Table Entries (PTE). The PTE is designated using this base address and some more bits from the original virtual address. The PTE in turn points to the base address of the page in physical memory where the data is stored. The final address in physical memory is the base address of this page plus the remaining bits from the original virtual address. The least significant bit in a PDE or PTE entry is the Valid or V bit. When this bit is one the entry is said to be ‘valid’ and bits 12-31 of the entry contain the Page Frame Number (PFN) used in the next part of the address translation. In a PDE, the PFN points to the page containing the PTE. In a PTE, the PFN points to the page containing the memory indicated in the original virtual address. On the other hand, when the V bit is zero the entry is said to be ‘invalid’ and a different set of rules must be used to find the data in question.

1. Page Directories

Each process has one page directory; a page the memory manager uses to map the location of all page tables for that process. The physical address of the process page directory is stored in the kernel process (KPROCESS) block, but it is also mapped virtually at address 0xC0300000 on x86 systems. The CPU knows the location of the page directory page because a special register (CR3 on x86 systems) that is loaded by the memory manager contains the physical address of the page directory. Each time a thread begins execution that is not part of the current process this register is loaded with the address of the new process’ page directory. Context switches between threads in the same process don't cause the address to be reloaded because all threads within the same process share the same address space. The page directory is composed of page directory entries (PDEs), each of which is 4 bytes long and describes the state and location of all the possible page tables for that process. On x86 systems, 1024 page tables are required to represent the full 4-GB virtual address space. The process page directory that maps these page tables contains 1024 PDEs. Therefore, the page directory index needs to be 10 bits wide. Because Windows© provides a private address space for each process, each process has its own set of page tables to map its private address space. [5] However, the page tables that map system space are shared among all processes. To avoid having multiple page tables describing the same virtual memory, when a process is created, the page directory entries that describe system space are initialized to point to the existing system page tables.

2. Page Tables

The process page directory entries point to individual page tables. Page tables are composed of an array of PTEs. The virtual address's page table index field (as shown in Figure 4) indicates which PTE within the page table maps the data page in question. On x86 systems, the page table index is 10 bits wide, allowing you to reference up to 1024 4-byte PTEs. However, because 32-bit Windows© provides a 4-GB private virtual address space, more than one page table is needed to map the entire address space. To calculate the number of page tables required to map the entire 4-GB process virtual address space, divide 4 GB by the virtual memory mapped by a single page table. Recall that each page table on an x86 system maps 4 MB of data pages. Thus, 1024 page tables (4 GB/4 MB) are required to map the full 4-GB address space. Valid PTEs have two main fields: the page frame number (PFN) of the physical page containing the data or of the physical address of a page in memory, and some flags that describe the state and protection of the page, as shown in Figure 3.3.

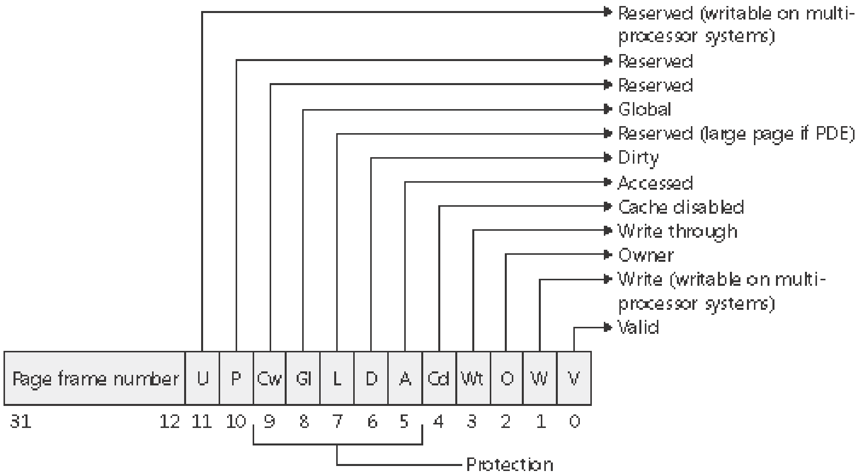

Figure 3.4 Valid x86 Hardware PTEs [From 5]

3. Individual Bytes

Once the memory manager has found the physical page in question, it must find the requested data within that page and this is where the byte index field comes in. The byte index field tells the CPU which byte of data in the page you want to reference. On x86 systems, the byte index is 12 bits wide, allowing you to reference up to 4096 bytes of data (the size of a page). So, adding the byte offset to the physical page number retrieved from the PTE completes the translation of a virtual address to a physical address.

B. THE PAGING SYSTEM

Ideally, when a process reads from, or writes to, a location in its virtual address space, that address is available in physical memory for immediate use. The working set is the set of virtual pages currently in main memory. But a system is not capable of providing enough physical addresses in main memory to map to every address of each process's virtual address space. When a thread references a virtual page that is not in the working set, a page fault occurs. The page fault triggers the memory manager to load the desired page into memory and update the working set to include the newly loaded virtual page. This is a valid page fault and is normally transparent to the user. An invalid page fault is a request for data that is not accessible and will usually result in a system crash.

If there is a need to free space in RAM, then parts of code and data that are not currently needed can be transferred to the hard disk in the page-file. This file is generally called pagefile.sys and is usually stored in the root directory of the primary drive; however one should check the system settings for its actual location. The format of the page-file is very simple, and can be treated the same as memory. It is simply a contiguous block of data which can be referenced by the same offsets as used for physical memory. Because of the constant swapping of pages, the total address space of virtual memory is rarely contained entirely within physical memory, nor is it ever constant.

Only the parts of the program and data that are in active use need to be in physical memory. Thus, parsing and analyzing the contents of a RAM dump in isolation without the page-file will not provide a complete view of memory. This is because pages that have been swapped out to the page-file are not utilized in the memory analysis. To overcome this Nicholas Maclean published his thesis work, Acquisition and Analysis of Windows Memory [13], in the spring of 2006. He explains the inner workings of the Windows© memory management system and provides an open-source tool written in Python called vtop to reconstruct the virtual address space of a process. Jesse Kornblum also released his work referred to as the “Buffalo” paper or by its full title, “Using Every Part of the Buffalo in Windows Memory Analysis” early in 2007 [8]. In this paper, Jesse demonstrates how more data can be recovered using invalid Page Table Entries.

C. INVALID ENTRIES

Just because an entry is invalid, doesn’t mean that the data it references is inaccessible. [8] One can follow the same rules as the operating system to access the data in question; however, it is still possible that the data was never loaded into memory and thus is truly inaccessible. When a page is removed from memory it is marked as invalid. An invalid PTE or PDE will always have its least-significant bit set to zero, but there are a number of reasons for it being invalid each of which uses a slightly different format and each invalid PDE or PTE fits into one of six categories: Page-file, Demand Zero, Transition, Prototype, Zero, or Unknown. Below we describe each of these in detail.

1. Page-File

If both the P and T bits in an invalid PTE or PDE entry are zero, the entry points to a frame in one of the paging files. Windows© can support up to 16 paging files, so the page-file number, PageFileNumber, is given in bits 1-4. The offset of the desired frame in the page-file, PageFileOffset, is in bits 12-31 of the invalid entry. [8] The true offset in the paging file is the value of bits 12-31 from the entry plus some bits from the original virtual address. Note that both PDEs and PTEs can point into the Page-file. For a PDE Page-file entry, PageFileOffset uses bits 12-21, shifted right 12 places, from the original virtual address being referenced. For a Page-file PTE entry, PageFileOffset uses bits 0-11 from the original virtual address.

2. Demand Zero

Like a page-file entry, Demand Zero entries have zeros in the T and P bits. But when the PageFileNumber and PageFileOffset are both zero, the operating system has marked the requested page as Demand Zero and would return any request for it with a page of zeros.

3. Transition

When the T bit in an entry is one and the P bit is zero, the page is said to be in Transition. [8] This means that the page has been modified but not yet written back to the disk. Even though a page was in transition, the page was still in active memory and can therefore be retrieved by an examiner. Just like a valid entry, the page frame number is given in bits 12-31 and can be used to continue the address translation process.

4. Prototype

In a PTE, when the P bit is one, the entry is a pointer to a prototype page table entry. Note that when P is one, the value of the T bit is part of the prototype's index. The entry contains an index number that can be used to compute the virtual address of the prototype PTE. Prototype PTEs are used when more than one process is using the same page in memory. [8] Prototypes are created when the operating system needs to invalidate the page in question. This is to avoid having to update all of the processes using the page each time the page is moved. Instead, they direct each process using the page to point to the same prototype. The prototype then points to the page's true location. When a page in question is moved, the memory manager only has to update the one prototype. The PTE stored by each process acts like a shortcut or symbolic link to the true PTE. Each Prototype PTE should be in one of the six states listed below.

- Active: The V bit is one. The page was in memory and can be accessed using the Page Frame Number in the prototype PTE entry.
- Transition: The V bit is zero and the T bit is one. The page was in Transition, but can be accessed in the memory image using the Page Frame Number in the prototype PTE entry.

- Modified No-Write: Like a transition prototype PTE, but the Dirty bit, bit six, is also one. The page can be accessed in the memory image using the Page Frame Number in the prototype PTE entry.
- Page File: The V, T, and P bits are all zero. The data are stored in the page file.
- Demand Zero: The V, T, and P bits are zero along with the PageFileNumber and PageFileOffset. The page should be satisfied with all zeros.
- Mapped File: The P bit is one. The operating system would retrieve the requested data from the original file on the disk. The author does not know how to use the value from these prototype PTEs. [8]

5. Zero

If the entry is zero, there is no information available for the page in question. Specifically, the page has been committed, but has not yet been accessed. That is, the operating system has allocated the page but has not read from or written to it. In this situation, the page does not contain any information relating to the process of which it belongs. However, it may still contain residual information from a process that has released the page for reuse by another process. Some processes will zero out a page that is allocated or fill it with random data; but this is not always the case. If these pages do contain data from another process the investigator will not be able to tell where the data came from without more information which can be obtained through a more in-depth analysis of the memory image.

6. Unknown

It is still possible that the status of the address is unknown and in this case the entries are put in this category and disregarded. [8]

Until recently, the only investigation performed on physical memory was similar to the analysis performed by system administrators on crash dumps to find out the cause for a crash. Theoretically, the contents of physical memory can show everything about the state of the system at the time the memory dump was taken (open files, running processes, network connections, logged on users). However, the analysis of the complete image of physical memory and page-file is able to reveal much more information. [8] When a process terminates, the pages of memory which were allocated to it are marked as free by the memory management system, but the actual contents of these pages are not overwritten until the memory is used. This means that this portion of memory actually contains data which is logically no longer part of the system. [9] This data may prove invaluable to a forensic investigation as it will be totally invisible to conventional crash dump analysis and the only way to recover it, is by means of an in depth forensic analysis of physical memory. [9]

IV. PROCESS BASICS

Windows stores information about each process in an EPROCESS kernel structure. The EPROCESS structures for all processes are stored in the address space of the System process. Once the base address of the EPROCESS block is known, values can be read by adding that offset and reading the value. Of the 4GB virtual address space allocated to the process, only the 1st 2GB of the virtual address space are available for the process to write to, with the 2nd 2GB used as shared system memory (although this can be changed to 3GB for the process and 1GB for the system using a boot option) [5]. This means that by looking up all virtual addresses ranging from 0x00000000 to 0x80000000 (0-2GB) all memory writable by that process can be produced.

A. EPROCESS STRUCTURE

The executive process (EPROCESS) block is the most important structure in memory for forensic analysis because it is the starting point of any further investigation of that process. All EPROCESS blocks are part of a doubly linked list, which includes blocks for active processes, although it is not uncommon to encounter exited processes in the list. This happens when an exited process is still opened as a handle by another active process; an exited EPROCESS block is deleted only when the last handle to the process is closed.

Each process on a Windows system is represented as an executive process, or EPROCESS, block. This EPROCESS block is a data structure in which various attributes of the process, as well as pointers to a number of other attributes and data structures relating to the process, are maintained. Because the data structure is a sequence of bytes, each sequence with a specific meaning and purpose, these structures can be read and analyzed by an investigator.

B. EXECUTIVE THREAD

Threads are entities within a process that represent executable code of the process. Every process starts with a single thread, which may spawn other threads. Multiple threads allow a process to do several tasks in parallel, with each thread doing a separate task. This is also more efficient because when the Operating System switches threads in the same process it does not cause a VM context switch.

Windows represents a thread by the executive thread (ETHREAD). Threads are important to forensic analysis because threads are scheduled for execution not processes. A thread's creation and exit times are stored in its ETHREAD block. The initial execution address for a thread is also stored in the ETHREAD block.

C. SEARCHING FOR PROCESSES

Now the problem is to find a pattern which reliably identifies the objects. This section describes the data structures representing processes and threads. It notes constant values and formulates rules which build upon pairs of offsets and values. [14] Other rules will be derived from functional requirements of the operating system.

Below we describe an algorithm for a scanner that will identify process and thread objects. [14] The scanner advances through the whole dump file at a step width equal to the kernel's memory allocation granularity. At every position the scanner looks for a valid process and thread structure by reading in the data and parsing it. This object is then evaluated based on a rule set.

1. Constraints

Values used to find valid process structures: [14]

Pcb.Header.Type != 0x03

Pcb.Header.Size != 0x1b

Pcb.DirectoryTableBase[0] == 0

Pcb.DirectoryTableBase[0] % 0x1000) != 0

Pcb.ThreadListHead.Flink < KERNEL_OFFSET

Pcb.ThreadListHead.Blink < KERNEL_OFFSET

WorkingSetLock.Event.Type != 0x01

WorkingSetLock.Event.Size != 0x04

V. SOFTWARE IMPLEMENTATION

A. IMPLEMENTATION

The scanner was implemented in C using Notepad as the text editor and gcc as the compiler. It has been tested in cygwin^{tm} [21] running on Windows$^{©}$ XP SP2. The system we developed and tested with is an Intel P-III 1.1 Ghz laptop with 512 MB of RAM. We needed no flags or modules other than those listed in the header file in Appendix A. Simply compile using a standard gcc command with the header located in the same directory as the source file.

B. USAGE

Below we provide a reprint of the "usage" string printed by the scanner we implanted.

```
Usage statement:  ./eproc.exe option <imageFile>

-l Walk active process list: ./eproc.exe -l <imageFile> <pAddress>

-s Scan image for process structures:  ./eproc.exe -s <imageFile>

-t Translates Virtual Address: ./eproc.exe -t <imageFile> <pAddress>
<vAddress>

-d Dump process information:  ./eproc.exe -d <imageFile> <pAddress>

-i Identify Operating System:  ./eproc.exe -i <imageFile> -verbose

-p Gets the number of pages in memory : ./eproc.exe -p <imageFile>
<pAddress>

-m Dump Process Memory:  ./eproc.exe -l <imageFile> <pAddress>
<pageFile>
```

C. FEATURES

1. Walk Active Process List

In conducting an investigation, after the preliminary data collection is complete the investigator will start the off-line analysis. Using this tool the first thing that the investigator should do is to get a process listing using the active process list. The command line to do this would look like: `./eproc.exe -l <imageFile>` It will search for the System process, which is the active process head, starting at the beginning of the file, using the same mechanism that is used to scan the image for process structures and will traverse the list once it is found. Starting at the active process list head, System process on Windows, the tool follows the forward links (Flinks) and backward links (Blinks). They are pointers to the process before or after the current process in the active process list. Optionally if the physical address of the System process is known it can be entered at the command line after the path to the image file and the program will begin its search at that address. The closer you get the less time it will take to find the system process.

One weakness with this method is that it will not find processes that have been removed from the active process list by DKOM (Direct Kernel Object Manipulation) [17]. Such processes can be hidden because a device driver or loadable kernel module has access to kernel memory and can modify objects in the kernel memory space in a reliable fashion in order to hide objects such as processes. In order to hide a process using object manipulation you must first locate the EPROCESS block of the process that you want to hide, then change the process behind it in the list to point to the process after the process you are trying to hid. Finally, change the process after it to point to the process before the one you are trying to hide. [17] Now, the active process list points around the hidden process and it is not reported when the system returns the active process list.

The process continues to run because scheduling in Windows© is thread based and not process based. [17] So while the process is not reported by the Operating System, it continues to schedule the processes threads for execution without adverse affect to the system. [17] However, even though this function may not list all of the processes that could be running on the system it is a good initial listing and will show processes that have not been unlinked and still may look suspicious, but are not sophisticated enough to remove themselves from the process list. It also creates a baseline to compare the results from a full memory scan. Any discrepancies between the listing and the scan indicate areas that could use further investigation.

2. Scan Image for Process Structures

Once the investigator has a listing of the active processes the next step is to scan the image for process structures. This is the mechanism that is used to find the active process head when walking the list; however the list function stops scanning once the System process is found and uses the linked list to find processes. The following command will allow the investigator to scan a memory image: `./eproc.exe -s <imageFile>` The scan may take several minutes to complete depending on the system on the size of the memory image. The results of the scan provide the investigator with a list of every block of data that fits the process structure using expected values in specific locations. The algorithm scans the memory image at an 8-byte interval because processes are 8-byte aligned and compares each chunk of data to a process structure. If values in specific fields match then a valid process is found.

The advantage to this is that it will find all process structures that are in the memory image regardless of whether they have been removed through DKOM or have been recently exited as long as the structure is still in memory and still meets the criteria that describe a valid process structure. The disadvantage to this method of gaining process information is that it is slow and on large images will take significantly longer than walking the list if you need quick results. Comparing the results of both a listing and scan will provide the best results for investigators and give them a reference point to direct further investigation.

3. Information Provided from Process Lists

The useful information provided from the list and scan functions are the image name and size which are provided in order to provide descriptive information to the investigator for future reference:

- Each process name, which can sometimes provide the investigator an idea as to the function of the process and whether it is a process that is suspect or not,
- Pid and PPid give information as to which process spawned the current process which is useful in determining where the processes originated,

- Time created can be used to determine a timeline of events in an investigation,
- Offset in image is important for further investigation because this is necessary input for the functions that work on specific processes,
- Page directory base address this is also useful information for manual analysis and is used by the tool for address translation.

The information provided in the listing provides broad overview of what was running on the system and can give clues as to where to investigate further.

4. Translates Virtual to Physical Address

Another feature is the ability of the program to translate a virtual address to a physical offset in to the memory image. A command that will achieve this is: `./eproc.exe -t <imageFile> <pAddress> <vAddress>` The examiner may want to use this feature to aid in any manual analysis of the memory image. The procedure for this is to enter the flag then the address to the process of which the address is taken then the address to be translated. As of now it does not work for addresses that have been paged.

5. Dump Process Information

Now that the investigator has an idea of which processes need further investigation, the next step is to get the process image data from memory. This function takes the offset to a process structure as its argument and overlays a process structure to the data. An example command is: `./eproc.exe -d <imageFile> <pAddress>` If the data found at the address provided fits the valid process format it prints information about the process including its DOS Header, File Header, Optional Header, and Section data. The information provided by this function is most useful when combined with a manual analysis on the image. One feature under development is the ability to carve out the data referenced in the headers and specific sections for further analysis. If there is not a valid process structure found at the offset given, the program prints: “no valid process found”.

6. Identify Operating System

Normally the investigator will already know the Operating System the image was taken from, but it may be useful to identify the Operating System for verification. A command to identify the Operating System of an image is: `./eproc.exe -i <imageFile> -verbose` Also, the verbose option returns information about the Kernel image that may prove to be useful in the investigation. This function will print out the version of windows that the image was captured from. It does this by locating the Kernel Base Address and comparing it to addresses known to be used by common Operating Systems [14].

7. Gets the Number of Pages in Memory

Statistics relating to the number of valid and invalid pages referenced in the page directory is useful in determining memory usage and if a process was exhibiting suspect behavior. The command to get the memory usage of a process is: `./eproc.exe -p <imageFile> <pAddress> to` get these statistics the scanner simply counts the number of valid and invalid addresses are in the page table for the specified process. Once a valid process is found, the page directory can be found using the process structures provided in appendix A. With the location of the page directory the rest of the virtual to physical translation information is found using the proper translation method.

8. Dump Process Memory

Now that the investigator has found the process that is suspect the next step is to retrieve its memory space from the memory image and the page file. To do this the command would be: `./eproc.exe -l <imageFile> <pAddress> <pageFile>` this function takes the image file and optionally a page-file and dumps the processes memory space to a file in its own folder. The file and folder will be named with the process name and the PID (process id). In order to reduce the amount of space required it does not dump global pages from the system shared memory space. The algorithm used to find the data and output it to a file goes through all of the virtual addresses in the page tables and translates them using ether the valid address translation method or the invalid address translation method. For invalid addresses if the address flags indicate it is in the page-file it retrieves the page from the page-file if it is provided. The output function simply writes the pages to the output file in the order they appear in the page tables. This puts the pages in ascending order based on the virtual address of the page. A second file is produced that contains the virtual addresses listed in the same order as the dump file so that they can be later matched to the corresponding page from memory. A useful function would be a mechanism to search the dumped file and would be useful and aid the investigator in advanced analysis. This would make it easier to reconstruct memory space and look at specific parts that may need further investigation. One feature that is being explored is the ability to find and dump specific parts of the process image, such as specific DLLs or all or part of the process executable.

D. TESTING

This program was tested using memory captured dd.exe. The system was a Laptop running Windows XP Service Pack 2 with 512 MB of RAM. The page-file was obtained by shutting down the system by removing the batteries and booting the system with Backtrack [22] to allow access and to ensure the integrity of the file. The results of the test runs can be found in Appendix C.

VI. SUMMARY

This report has shown that during the forensic process, as much attention must be paid to volatile memory as to the more traditional sources of evidence. It should no longer be standard practice for the contents of memory to be destroyed in the interest of the integrity of the data on the hard disks. Different techniques are either available now or will be available in the near future for acquiring memory images. Once a memory image has been taken it is possible to locate pages that would not normally be available to the investigator by incorporating information in the process's page directories. Currently this tool retrieves the pages in the order they appear in the page directories, it would be better to sort them in order of virtual address, however this is still in progress. Another feature under development is the ability to analyze the data it produces. There are few tools currently available which can analyze this data, however, it can be done manually and automated tools are under development. The release of the *Volatility* framework will hopefully allow developers to add to this toolbox easily. It is expected that in the near future, full suites of memory analysis software will be available, and the investigation of memory will be viewed with the same importance as any other areas of computer forensics.

A. FUTURE WORK

This paper has attempted to expand the amount of information available to an examiner conducting Windows© memory analysis. The current implementation of this tool extracts process data in its entirety and saves it in one consolidated file. One area that needs to be addressed is the ability to extract specific parts of a process such as only the executable or specific DLLs. This would facilitate an investigator's search and allow more time for analysis. Another area that needs further development is the robustness of options available that will allow the user to more easily conduct data retrieval and analysis. Although demonstrating that more information is available when using robust address translation, there are still many opportunities to increase the amount of recoverable data in a memory image. This will hopefully allow for more advanced programs to be produced which could, for example analyze data structures and variables within process space and produce visualization systems for displaying the contents of memory in readable ways.

There is also much more information available to the investigator by looking for other types of data structures. Future developments will be able to find loaded DLLs, open file handles and which thread is accessing the files, encryption keys, and open sockets just to name a few. All of this information will be useful to the investigator in recreating the scene of the digital crime. Knowing if there were open sockets, threads executing, and which files were being accessed at the time of the incident give the investigator invaluable information as to the type, method, and extent of damage that may have occurred. All of this information is available to some extent, but to take full advantage of the information it provides it needs to be organized and analyzed in meaningful ways.

Hibernation may prove to be a useful method of acquiring memory, but more research needs to be done to make this a reliable tool in the forensic investigator's tool box. The ways in which Windows fetches, compresses, and stores memory are not fully understood, and it is not clear if any other operations are performed during the hibernation process. Researchers have had some success in reverse engineering the proprietary algorithm used to hibernate systems running Windows©, however there is not enough evidence to show that the methods they use yield the results required by forensic analysis. Furthermore, this method requires that hibernation be enabled prior to the incident which limits it usefulness to certain cases. Another consideration is that hibernation a process operated by the system that is in question and it may be compromised possible yielding unreliable results. If the system has been compromised nothing running on the system can be trusted.

Firewire has been designed to access memory through DMA, which should lend it towards memory acquisition with minimal impact on the target system. Research has shown that this is possible on Windows© XP systems, but has yet to produce a viable solution which would be suitable for incident response. While in some cases this may be an acceptable method, the investigator must introduce new hardware to the system thus changing its state. Whenever the investigator changes the state of the system by introducing new hardware, one cannot rule out the possibility that it will be recognized alerting the intruder and allowing for the destruction of evidence.

A multitude of applications relating to and stemming from this research can be found in incident response, data recovery, computer forensics, and cyber crime investigation. These are all interrelated, but each one is its own independent research area with unique difficulties and opportunities. The field of memory analysis can be used to determine what was happening on a system at the time the memory image was taken, to find encryption keys and possibly reconstruct deleted files. The ultimate result of research is to completely reconstruct the state of a live system from a memory image and hard disk. The ability to do this is like being able to freeze a crime scene and completely reconstruct it in a controlled environment, possibly even while the criminal is still present. Further research will lead to greater understanding of computer systems leading to tools that will aid solving cyber crimes and convicting suspects.

The final area of future research and possibly the most important when it comes to catching and convicting criminals is the admissibility of evidence obtained in the Court of Law. Once the investigator has used the information to solve the crime it must then be used in court to convict the suspect. As of now, the tool is simply an evidence acquisition tool used to diagnose symptoms of a system that appears to be compromised and only a small piece of the overall investigation. However, in the future there will have to be some way to prove that the dump came from the system in question, that it has not been compromised, and be able to provide conclusive evidence that a crime was committed and that the suspect did indeed commit the crime in question. Not only must the tools used by an investigator do this, but it must do it in proven, repeatable ways.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. WINDOWS STRUCTURES

A. WIN32STRUCTS.H

```
/*********************************************************************
Name Windows Physical Memory Offline Analyzer [7]
File win32structs.h
Version 1.0
Author John Schultz

Description Console Program which enumerates the processes running on a
             Windows PC. The program reads an offline copy of the
             physical memory for all the information gathered about the
             running processes.

*********************************************************************/

#include<sys/cdefs.h>

#ifndef __WIN32STRUCTS_H
#define __WIN32STRUCTS_H

__BEGIN_DECLS

/*Each struct is defined based on that given by the Windows(R) Kernel
Debugger with few exceptions, which are noted*/
struct _unnamed {
    /* +0x000 */ long double var01;
    /* +0x00a */ long double var02;
    /* +0x014 */ long double var03;
    /* +0x01e */ long double var04;
};

struct LIST_ENTRY {
    /* +0x000 */ void *Flink;
    /* +0x004 */ void *Blink;
};

struct UNICODE_STRING {
    /* 0x000 */ unsigned short  Length;
    /* 0x002 */ unsigned short  MaximumLength;
    /* 0x004 */ unsigned short *Buffer;
};

struct STRING {
    /* 0x000 */ unsigned short Length;
    /* 0x002 */ unsigned short MaximumLength;
    /* 0x004 */ char          *Buffer;
};

struct DISPATCHER_HEADER {
    /* +0x000 */ unsigned char     Type;
    /* +0x001 */ unsigned char     Absolute;
    /* +0x002 */ unsigned char     Size;
    /* +0x003 */ unsigned char     Inserted;
    /* +0x004 */ unsigned long     SignalState;
    /* +0x008 */ struct LIST_ENTRY WaitList;
};

struct KEVENT {
    /* +0x000 */ struct DISPATCHER_HEADER Header;
};

struct KDEVICE_QUEUE {
    /* +0x000 */ short              Type;
    /* +0x002 */ short              Size;
    /* +0x004 */ struct LIST_ENTRY  DeviceListHead;
    /* +0x00c */ unsigned long      Lock;
    /* +0x010 */ unsigned char      Busy;
};

struct KDPC {
    /* +0x000 */ short             Type;
    /* +0x002 */ unsigned char     Number;
    /* +0x003 */ unsigned char     Importance;
    /* +0x004 */ struct LIST_ENTRY DpcListEntry;
    /* +0x00c */ void             *DeferredRoutine;
    /* +0x010 */ void             *DeferredContext;
    /* +0x014 */ void             *SystemArgument1;
    /* +0x018 */ void             *SystemArgument2;
    /* +0x01c */ unsigned long    *Lock;
};

struct MMWSLE_HASH {
  /* +0x000 */ void         *Key;
  /* +0x004 */ unsigned long Index;
};

struct MMSUPPORT_FLAGS {
  /* +0x000 */ int Sessionspace : 1;
  /* +0x000 */ int BeingTrimmed : 1;
  /* +0x000 */ int SessionLoader : 1;
  /* +0x000 */ int TrimHard : 1;
  /* +0x000 */ int WorkingSetHard : 1;
  /* +0x000 */ int AddressSpaceBeingDeleted : 1;
  /* +0x000 */ int Available : 10;
  /* +0x000 */ int AllowWorkingSetAdjustment : 8;
  /* +0x000 */ int MemoryPriority : 8;
};

struct MMWSL {
  /* +0x000 */ unsigned long       Quota;
  /* +0x004 */ unsigned long       FirstFree;
  /* +0x008 */ unsigned long       FirstDynamic;
  /* +0x00c */ unsigned long       LastEntry;
  /* +0x010 */ unsigned long       NextSlot;
  /* +0x014 */ void               *Wsle; /*points to MMWSLE*/
  /* +0x018 */ unsigned long       LastInitializedWsle;
  /* +0x01c */ unsigned long       NonDirectCount;
  /* +0x020 */ struct MMWSLE_HASH *HashTable;
  /* +0x024 */ unsigned long       HashTableSize;
  /* +0x028 */ unsigned long       NumberOfCommittedPageTables;
  /* +0x02c */ void               *HashTableStart;
  /* +0x030 */ void               *HighestPermittedHashAddress;
  /* +0x034 */ unsigned long       NumberOfImageWaiters;
  /* +0x038 */ unsigned long       VadBitMapHint;
  /* +0x03c */ unsigned short      UsedPageTableEntries[768];
  /* +0x63c */ unsigned long       CommittedPageTables[24];
};

struct MMSUPPORT {
  /* +0x000 */ long long              LastTrimTime;
  /* +0x008 */ struct MMSUPPORT_FLAGS Flags;
  /* +0x00c */ unsigned long          PageFaultCount;
  /* +0x010 */ unsigned long          PeakWorkingSetSize;
  /* +0x014 */ unsigned long          WorkingSetSize;
  /* +0x018 */ unsigned long          MinimumWorkingSetSize;
  /* +0x01c */ unsigned long          MaximumWorkingSetSize;
  /* +0x020 */ struct MMWSL          *VmWorkingSetList;
  /* +0x024 */ struct LIST_ENTRY      WorkingSetExpansionLinks;
  /* +0x02c */ unsigned long          Claim;
  /* +0x030 */ unsigned long          NextEstimationSlot;
  /* +0x034 */ unsigned long          NextAgingSlot;
  /* +0x038 */ unsigned long          EstimatedAvailable;
  /* +0x03c */ unsigned long          GrowthSinceLastEstimate;
};

struct FAST_MUTEX {
    /* +0x000 */ long                     Count;
    /* +0x004 */ void                    *Owner;
    /* +0x008 */ unsigned long            Contention;
    /* +0x00c */ struct DISPATCHER_HEADER Event;
    /* +0x01c */ unsigned long            OldIrql;
};

struct CURDIR {
    /* 0x000 */ struct UNICODE_STRING DosPath;
    /* 0x008 */ void                 *Handle;
};

struct PEB_LDR_DATA {
    /* 0x000 */ unsigned long     Length;
    /* 0x004 */ unsigned char     Initialized;
    /* 0x008 */ void             *SsHandle;
    /* 0x00c */ struct LIST_ENTRY InLoadOrderModuleList;
    /* 0x014 */ struct LIST_ENTRY InMemoryOrderModuleList;
    /* 0x01c */ struct LIST_ENTRY InInitializationOrderModuleList;
    /* 0x024 */ void             *EntryInProgress;
};

struct KSEMAPHORE {
    /* +0x000 */ struct DISPATCHER_HEADER Header;
    /* +0x010 */ long   Limit;
};

struct KGDTENTRY {
    /* +0x000 */ unsigned short    LimitLow;
    /* +0x002 */ unsigned short    BaseLow;
    /* +0x004 */ unsigned long     HighWord;
};

struct KIDTENTRY {
    /* +0x000 */ unsigned short    Offset;
    /* +0x002 */ unsigned short    Selector;
    /* +0x004 */ unsigned short    Access;
    /* +0x006 */ unsigned short    ExtendedOffset;
};

struct HANDLE_TABLE {
    /* +0x000 */ unsigned long     TableCode;
    /* +0x004 */ struct EPROCESS  *QuotaProcess;
    /* +0x008 */ void             *UniqueProcessID;
    /* +0x00c */ unsigned long     HandleTableLock[4];
    /* +0x01c */ struct LIST_ENTRY HandleTableList;
    /* +0x02c */ unsigned long     HandleContentionEvent;
    /* +0x030 */ void             *DebugInfo;
    /* +0x034 */ long              ExtraInfoPages;
    /* +0x038 */ unsigned long     FirstFree;
    /* +0x03c */ unsigned long     LastFree;
    /* +0x040 */ unsigned long     NextHandleNeedingPool;
    /* +0x044 */ long              HandleCount;
    /* +0x048 */ unsigned long     Flags;
};

struct OWNER_ENTRY {
    /* +0x000 */ unsigned long OwnerThread;
    /* +0x004    long          OwnerCount; */
    /* +0x004 */ unsigned long TableSize;
};

struct OBJECT_TYPE_INITIALIZER {
    /* +0x000 */ unsigned short  Length;
    /* +0x002 */ unsigned char   UseDefaultObject;
    /* +0x003 */ unsigned char   CaseInsensitive;
    /* +0x004 */ unsigned long   InvalidAttributes;
    /* +0x008 */ void           *GenericMapping;
    /* +0x018 */ unsigned long   ValidAccessMask;
    /* +0x01c */ unsigned char   SecurityRequired;
    /* +0x01d */ unsigned char   MaintainHandleCount;
    /* +0x01e */ unsigned char   MaintainTypeList;
    /* +0x020 */ void           *PoolType;
    /* +0x024 */ unsigned long   DefaultPagedPoolCharge;
    /* +0x028 */ unsigned long   DefaultNonPagedPoolCharge;
    /* +0x02c */ void           *DumpProcedure;
    /* +0x030 */ void           *OpenProcedure;
    /* +0x034 */ void           *CloseProcedure;
    /* +0x038 */ void           *DeleteProcedure;
    /* +0x03c */ void           *ParseProcedure;
    /* +0x040 */ void           *SecurityProcedure;
    /* +0x044 */ void           *QueryNameProcedure;
    /* +0x048 */ void           *OkayToCloseProcedure;
};

struct ERESOURCE {
    /* +0x000 */ struct LIST_ENTRY         SystemResourcesList;
    /* +0x008 */ struct OWNER_ENTRY       *OwnerTable;
    /* +0x00c */ short                     ActiveCount;
    /* +0x00e */ unsigned short            Flag;
    /* +0x010 */ struct KSEMAPHORE        *SharedWaiters;
    /* +0x014 */ struct DISPATCHER_HEADER *ExclusiveWaiters;
    /* +0x018 */ struct OWNER_ENTRY        OwnerThreads[2];
    /* +0x028 */ unsigned long             ContentionCount;
    /* +0x02c */ unsigned short            NumberOfSharedWaiters;
    /* +0x02e */ unsigned short            NumberOfExclusiveWaiters;
    /* +0x030    void                     *Address;*/
    /* +0x030 */ unsigned long             CreatorBackTraceIndex;
    /* +0x034 */ unsigned long             SpinLock;
};

struct SEGMENT {
    /* +0x000 */ struct CONTROL_AREA *ControlArea;
    /* +0x004 */ unsigned long        TotalNumberOfPtes;
    /* +0x008 */ unsigned long        NonExtendedPtes;
    /* +0x00c */ unsigned long        WritableUserReferences;
    /* +0x010 */ unsigned long long   SizeOfSegment;
    /* +0x018 */ long                 SegmentPteTemplate;
    /* +0x01c */ unsigned long        NumberOfCommittedPages;
    /* +0x020 */ /* struct MMEXTEND_INFO */void *ExtendInfo;
    /* +0x024 */ void                *SystemImageBase;
    /* +0x028 */ void                *BasedAddress;
    /* +0x02c */ long                 u1;
    /* +0x030 */ long                 u2;
    /* +0x034 */ long                *PrototypePte;
    /* +0x038 */ long                 ThePtes[1];
};

struct CONTROL_AREA {
    /* +0x000 */ struct SEGMENT       *Segment;
    /* +0x004 */ struct LIST_ENTRY     DereferenceList;
    /* +0x00c */ unsigned long         NumberOfSectionReferences;
    /* +0x010 */ unsigned long         NumberOfPfnReferences;
    /* +0x014 */ unsigned long         NumberOfMappedViews;
    /* +0x018 */ unsigned short        NumberOfSubsections;
    /* +0x01a */ unsigned short        FlushInProgressCount;
    /* +0x01c */ unsigned long         NumberOfUserReferences;
    /* +0x020 */ long                  unnamed;
    /* +0x024 */ struct FILE_OBJECT   *FilePointer;
    /* +0x028 */ /*struct EVENT_COUNTER*/void *WaitingForDeletion;
    /* +0x02c */ unsigned short        ModifiedWriteCount;
    /* +0x02e */ unsigned short        NumberOfSystemCacheViews;
};

struct SUBSECTION {
    /* +0x000 */ struct CONTROL_AREA  *ControlArea;
    /* +0x004 */ long                  unnamed;
    /* +0x008 */ unsigned long         StartingSector;
    /* +0x00c */ unsigned long         NumberOfFullSectors;
    /* +0x010 */ /*struct MMPTE*/void         *SubsectionBase;
    /* +0x014 */ unsigned long         UnusedPtes;
    /* +0x018 */ unsigned long         PtesInSubsection;
    /* +0x01c */ struct SUBSECTION    *NextSubsection;
};
```

```
struct SEGMENT_OBJECT {
    /* +0x000 */ void                     *BaseAddress;
    /* +0x004 */ unsigned long             TotalNumberOfPtes;
    /* +0x008 */ long long                 SizeOfSegment;
    /* +0x010 */ unsigned long             NonExtendedPtes;
    /* +0x014 */ unsigned long             ImageCommitment;
    /* +0x018 */ struct CONTROL_AREA      *ControlArea;
    /* +0x01c */ struct SUBSECTION        *Subsection;
    /* +0x020 */ /*struct LARGE_CONTROL_AREA*/void *LargeControlArea;
    /* +0x024 */ /*struct MMSECTION_FLAGS*/ void     *MmSectionFlags;
    /* +0x028 */ /*struct MMSUBSECTION_FLAGS*/ void
*MmSubSectionFlags;
};

struct SECTION_OBJECT_POINTERS {
    /* +0x000 */   void                  *DataSectionObject;
    /* +0x004 */   void                  *SharedCacheMap;
    /* +0x008 */   void                  *ImageSectionObject;
};

struct SECTION_OBJECT {
    /* +0x000 */   void                  *StartingVa;
    /* +0x004 */   void                  *EndingVa;
    /* +0x008 */   void                  *Parent;
    /* +0x00c */   void                  *LeftChild;
    /* +0x010 */   void                  *RightChild;
    /* +0x014 */   struct SEGMENT_OBJECT *Segment;
};

struct OBJECT_TYPE {
    /* +0x000 */ struct ERESOURCE                  Mutex;
    /* +0x038 */ struct LIST_ENTRY                 TypeList;
    /* +0x040 */ struct UNICODE_STRING             Name;
    /* +0x048 */ void                             *DefaultObject;
    /* +0x04c */ unsigned long                     Index;
    /* +0x050 */ unsigned long                    TotalNumberOfObjects;
    /* +0x054 */ unsigned long                    TotalNumberOfHandles;
    /* +0x058 */ unsigned long                HighWaterNumberOfObjects;
    /* +0x05c */ unsigned long                HighWaterNumberOfHandles;
    /* +0x060 */ struct OBJECT_TYPE_INITIALIZER    TypeInfo;
    /* +0x0ac */ unsigned long                     Key;
    /* +0x0b0 */ struct ERESOURCE                  ObjectLocks[4];
};

struct OBJECT_HEADER {
    /* +0x000 */ long                     PointerCount;
    /* +0x004    long                     HandleCount; */
    /* +0x004 */ void                    *NextToFree;
    /* +0x008 */ struct OBJECT_TYPE      *Type;
    /* +0x00c */ unsigned char            NameInfoOffset;
    /* +0x00d */ unsigned char            HandleInfoOffset;
    /* +0x00e */ unsigned char            QuotaInfoOffset;
    /* +0x00f */ unsigned char            Flags;
    /* +0x010 */ /* struct OBJECT_CREATE_INFORMATION *ObjectCreateInfo;
*/
    /* +0x010 */ void                    *QuotaBlockCharged;
    /* +0x014 */ void                    *SecurityDescriptor;
};

struct VPB {
    /* +0x000 */ short                        Type;
    /* +0x002 */ short                        Size;
    /* +0x004 */ unsigned short               Flags;
    /* +0x006 */ unsigned short               VolumeLabelLength;
    /* +0x008 */ struct DEVICE_OBJECT        *DeviceObject;
    /* +0x00c */ struct DEVICE_OBJECT        *RealDevice;
    /* +0x010 */ unsigned long                SerialNumber;
    /* +0x014 */ unsigned long                ReferenceCount;
    /* +0x018 */ unsigned short               VolumeLabel[32];
/*Unicode name*/
};

struct DEVICE_OBJECT {
    /* +0x000 */ short                    Type;
    /* +0x002 */ unsigned short           Size;
    /* +0x004 */ long                     ReferenceCount;
    /* +0x008 */ struct DRIVER_OBJECT    *DriverObject;
    /* +0x00c */ struct DEVICE_OBJECT    *NextDevice;
    /* +0x010 */ struct DEVICE_OBJECT    *AttachedDevice;
    /* +0x014 */ /* struct IRP */ void             *CurrentIrp;
    /* +0x018 */ /* struct IO_TIMER */ void        *Timer;
    /* +0x01c */ unsigned long            Flags;
    /* +0x020 */ unsigned long            Characteristics;
    /* +0x024 */ struct VPB              *Vpb;
    /* +0x028 */ void                    *DeviceExtension;
    /* +0x02c */ unsigned long            DeviceType;
    /* +0x030 */ char                     StackSize;
    /* +0x034 */ struct _unnamed          Queue;
    /* +0x05c */ unsigned long            AlignmentRequirement;
    /* +0x060 */ struct KDEVICE_QUEUE     DeviceQueue;
    /* +0x074 */ struct KDPC              Dpc;
    /* +0x094 */ unsigned long            ActiveThreadCount;
    /* +0x098 */ void                    *SecurityDescriptor;
    /* +0x09c */ struct KEVENT            DeviceLock;
    /* +0x0ac */ unsigned short           SectorSize;
    /* +0x0ae */ unsigned short           Spare1;
    /* +0x0b0 */ /* struct DEVOBJ_EXTENSION */ void
*DeviceObjectExtension;
    /* +0x0b4 */ void                    *Reserved;
};

struct DRIVER_OBJECT {
    /* +0x000 */ short                    Type;
    /* +0x002 */ short                    Size;
    /* +0x004 */ struct DEVICE_OBJECT    *DeviceObject;
    /* +0x008 */ unsigned long            Flags;
    /* +0x00c */ void                    *DriverStart;
    /* +0x010 */ unsigned long            DriverSize;
    /* +0x014 */ void                    *DriverSection;
    /* +0x018 */ /* struct DRIVER_EXTENSION */ void *DriverExtension;
    /* +0x01c */ struct UNICODE_STRING    DriverName;
    /* +0x024 */ struct UNICODE_STRING   *HardwareDatabase;
    /* +0x028 */ /* struct FAST_IO_DISPATCH */ void *FastIoDispatch;
    /* +0x02c */ void                    *DriverInit;
    /* +0x030 */ void                    *DriverStartIo;
    /* +0x034 */ void                    *DriverUnload;
    /* +0x038 */ void                    *MajorFunction[28];
};

struct FILE_OBJECT {
    /* +0x000 */ short                           Type;
    /* +0x002 */ short                           Size;
    /* +0x004 */ struct DEVICE_OBJECT           *DeviceObject;
    /* +0x008 */ struct VPB                     *Vpb;
    /* +0x00c */ void                           *FsContext;
    /* +0x010 */ void                           *FsContext2;
    /* +0x014 */ struct SECTION_OBJECT_POINTERS *SectionObjectPointer;
    /* +0x018 */ void                           *PrivateCacheMap;
    /* +0x01c */ long                            FinalStatus;
    /* +0x020 */ struct FILE_OBJECT             *RelatedFileObject;
    /* +0x024 */ unsigned char                   LockOperation;
    /* +0x025 */ unsigned char                   DeletePending;
    /* +0x026 */ unsigned char                   ReadAccess;
    /* +0x027 */ unsigned char                   WriteAccess;
    /* +0x028 */ unsigned char                   DeleteAccess;
    /* +0x029 */ unsigned char                   SharedRead;
    /* +0x02a */ unsigned char                   SharedWrite;
    /* +0x02b */ unsigned char                   SharedDelete;
    /* +0x02c */ unsigned long                   Flags;
    /* +0x030 */ struct UNICODE_STRING           FileName;
    /* +0x038 */ long long                       CurrentByteOffset;
    /* +0x040 */ unsigned long                   Waiters;
    /* +0x044 */ unsigned long                   Busy;
    /* +0x048 */ void                           *LastLock;
    /* +0x04c */ struct KEVENT                   Lock;
    /* +0x05c */ struct KEVENT                   Event;
    /* +0x06c */ /* struct IO_COMPLETION_CONTEXT */ void
*CompletionContext;
};

struct EX_PUSH_LOCK {
    /* +0x000 */ int           Waiting : 1; //LSB
    /* +0x000 */ int           Exclusive : 1;
    /* +0x000 */ int           Shared : 30;
    /* +0x000 */ unsigned long Value;
    /* +0x000 */ void         *Ptr;
};

struct OBJECT_DIRECTORY_ENTRY {
    /* +0x000 */ struct OBJECT_DIRECTORY_ENTRY *ChainLink;
    /* +0x004 */ void                          *Object;
};

struct OBJECT_DIRECTORY {
    /* +0x000 */ struct OBJECT_DIRECTORY_ENTRY *HashBuckets[37];
    /* +0x094 */ struct EX_PUSH_LOCK            Lock;
    /* +0x098 */ struct DEVICE_MAP             *DeviceMap;
    /* +0x09c */ unsigned long                  SessionId;
    /* +0x0a0 */ unsigned short                 Reserved;
    /* +0x0a2 */ unsigned short                 SymbolicLinkUsageCount;
```

```
};

struct DEVICE_MAP {
    /* +0x000 */ struct OBJECT_DIRECTORY       *DosDevicesDirectory;
    /* +0x004 */ struct OBJECT_DIRECTORY
*GlobalDosDevicesDirectory;
    /* +0x008 */ unsigned long                  ReferenceCount;
    /* +0x00c */ unsigned long                  DriveMap;
    /* +0x010 */ unsigned char                  DriveType[32];
};

struct RTL_DRIVE_LETTER_CURDIR {
    /* +0x000 */ unsigned short Flags;
    /* +0x002 */ unsigned short Length;
    /* +0x004 */ unsigned long  TimeStamp;
    /* +0x008 */ struct STRING  DosPath;
};

struct RTL_USER_PROCESS_PARAMETERS {
    /* +0x000 */ unsigned long                  MaximumLength;
    /* +0x004 */ unsigned long                  Length;
    /* +0x008 */ unsigned long                  Flags;
    /* +0x00c */ unsigned long                  DebugFlags;
    /* +0x010 */ void                          *ConsoleHandle;
    /* +0x014 */ unsigned long                  ConsoleFlags;
    /* +0x018 */ void                          *StandardInput;
    /* +0x01c */ void                          *StandardOutput;
    /* +0x020 */ void                          *StandardError;
    /* +0x024 */ struct CURDIR                  CurrentDirectory;
    /* +0x030 */ struct UNICODE_STRING          DllPath;
    /* +0x038 */ struct UNICODE_STRING          ImagePathName;
    /* +0x040 */ struct UNICODE_STRING          CommandLine;
    /* +0x048 */ void                          *Environment;
    /* +0x04c */ unsigned long                  StartingX;
    /* +0x050 */ unsigned long                  StartingY;
    /* +0x054 */ unsigned long                  CountX;
    /* +0x058 */ unsigned long                  CountY;
    /* +0x05c */ unsigned long                  CountCharsX;
    /* +0x060 */ unsigned long                  CountCharsY;
    /* +0x064 */ unsigned long                  FillAttribute;
    /* +0x068 */ unsigned long                  WindowFlags;
    /* +0x06c */ unsigned long                  ShowWindowFlags;
    /* +0x070 */ struct UNICODE_STRING          WindowTitle;
    /* +0x078 */ struct UNICODE_STRING          DesktopInfo;
    /* +0x080 */ struct UNICODE_STRING          ShellInfo;
    /* +0x088 */ struct UNICODE_STRING          RuntimeData;
    /* +0x090 */ struct RTL_DRIVE_LETTER_CURDIR CurrentDirectores[32];
};

struct PEB {
    /* +0x000 */ unsigned char        InheritedAddressSpace;
    /* +0x001 */ unsigned char        ReadImageFileExecOptions;
    /* +0x002 */ unsigned char        BeingDebugged;
    /* +0x003 */ unsigned char        SpareBool;
    /* +0x004 */ void                *Mutant;
    /* +0x008 */ void                *ImageBaseAddress;
    /* +0x00c */ struct PEB_LDR_DATA *Ldr;
    /* +0x010 */ struct RTL_USER_PROCESS_PARAMETERS *ProcessParameters;
    /* +0x014 */ void                *SubSystemData;
    /* +0x018 */ void                *ProcessHeap;
    /* +0x01c */ struct RTL_CRITICAL_SECTION *FastPebLock;
    /* +0x020 */ void                *FastPebLockRoutine;
    /* +0x024 */ void                *FastPebUnlockRoutine;
    /* +0x028 */ unsigned long        EnvironmentUpdateCount;
    /* +0x02c */ void                *KernelCallbackTable;
    /* +0x030 */ unsigned long        SystemReserved[1];
    /* +0x034 */ unsigned long        AtlThunkSList;
    /* +0x038 */ struct PEB_FREE_BLOCK *FreeList;
    /* +0x03c */ unsigned long        TlsExpansionCounter;
    /* +0x040 */ void                *TlsBitmap;
    /* +0x044 */ unsigned long        TlsBitmapBits[2];
    /* +0x04c */ void                *ReadOnlySharedMemoryBase;
    /* +0x050 */ void                *ReadOnlySharedMemoryHeap;
    /* +0x054 */ void               **ReadOnlyStaticServerData;
    /* +0x058 */ void                *AnsiCodePageData;
    /* +0x05c */ void                *OemCodePageData;
    /* +0x060 */ void                *UnicodeCaseTableData;
    /* +0x064 */ unsigned long        NumberOfProcessors;
    /* +0x068 */ unsigned long        NtGlobalFlag;
    /* +0x070 */ long long            CriticalSectionTimeout;
    /* +0x078 */ unsigned long        HeapSegmentReserve;
    /* +0x07c */ unsigned long        HeapSegmentCommit;
    /* +0x080 */ unsigned long        HeapDeCommitTotalFreeThreshold;
    /* +0x084 */ unsigned long        HeapDeCommitFreeBlockThreshold;
    /* +0x088 */ unsigned long        NumberOfHeaps;
    /* +0x08c */ unsigned long        MaximumNumberOfHeaps;
    /* +0x090 */ void               **ProcessHeaps;
    /* +0x094 */ void                *GdiSharedHandleTable;
    /* +0x098 */ void                *ProcessStarterHelper;
    /* +0x09c */ unsigned long        GdiDCAttributeList;
    /* +0x0a0 */ void                *LoaderLock;
    /* +0x0a4 */ unsigned long        OSMajorVersion;
    /* +0x0a8 */ unsigned long        OSMinorVersion;
    /* +0x0ac */ unsigned short      *OSBuildNumber;
    /* +0x0ae */ unsigned short      *OSCSDVersion;
    /* +0x0b0 */ unsigned long        OSPlatformId;
    /* +0x0b4 */ unsigned long        ImageSubsystem;
    /* +0x0b8 */ unsigned long        ImageSubsystemMajorVersion;
    /* +0x0bc */ unsigned long        ImageSubsystemMinorVersion;
    /* +0x0c0 */ unsigned long        ImageProcessAffinityMask;
    /* +0x0c4 */ unsigned long        GdiHandleBuffer[34];
    /* +0x14c */ void                *PostProcessInitRoutine;
    /* +0x150 */ void                *TlsExpansionBitmap;
    /* +0x154 */ unsigned long        TlsExpansionBitmapBits[32];
    /* +0x1d4 */ unsigned long        SessionId;
    /* +0x1d8 */ unsigned long long   AppCompatFlags;
    /* +0x1e0 */ unsigned long long   AppCompatFlagsUser;
    /* +0x1e8 */ void                *pShimData;
    /* +0x1ec */ void                *AppCompatInfo;
    /* +0x1f0 */ struct UNICODE_STRING CSDVersion;
    /* +0x1f8 */ void                *ActivationContextData;
    /* +0x1fc */ void                *ProcessAssemblyStorageMap;
    /* +0x200 */ void              *SystemDefaultActivationContextData;
    /* +0x204 */ void                *SystemAssemblyStorageMap;
```

```
    /* +0x208 */ unsigned long        MinimumStackCommit;
};

struct HARDWARE_PTE {
    /* +0x000 */ int Valid : 1; //LSB
    /* +0x000 */ int Write : 1;
    /* +0x000 */ int Owner : 1;
    /* +0x000 */ int WriteThrough : 1;
    /* +0x000 */ int CacheDisable : 1;
    /* +0x000 */ int Accessed : 1;
    /* +0x000 */ int Dirty : 1;
    /* +0x000 */ int LargePage : 1;
    /* +0x000 */ int Global : 1;
    /* +0x000 */ int CopyOnWrite : 1;
    /* +0x000 */ int Prototype : 1;
    /* +0x000 */ int Reserved : 1;
    /* +0x000 */ int PageFrameNumber : 20;
};

struct KPROCESS {
    /* +0x000 */ struct DISPATCHER_HEADER Header;
    /* +0x010 */ struct LIST_ENTRY        ProfileListHead;
    /* +0x018 */ unsigned long            DirectoryTableBase[2];
    /* +0x020 */ struct KGDTENTRY         LdtDescriptor;
    /* +0x028 */ struct KIDTENTRY         Int21Descriptor;
    /* +0x030 */ unsigned short           IopmOffset;
    /* +0x032 */ unsigned char            Iopl;
    /* +0x033 */ unsigned char            Unused;
    /* +0x034 */ unsigned long            ActiveProcessors;
    /* +0x038 */ unsigned long            KernelTime;
    /* +0x03c */ unsigned long            UserTime;
    /* +0x040 */ struct LIST_ENTRY        ReadyListHead;
    /* +0x048 */ void                    *SwapListEntry;
    /* +0x04c */ void                    *VdmTrapcHandler;
    /* +0x050 */ struct LIST_ENTRY        ThreadListHead;
    /* +0x058 */ unsigned long            ProcessLock;
    /* +0x05c */ unsigned long            Affinity;
    /* +0x060 */ unsigned short           StackCount;
    /* +0x062 */ char                     BasePriority;
    /* +0x063 */ char                     ThreadQuantum;
    /* +0x064 */ unsigned char            AutoAlignment;
    /* +0x065 */ unsigned char            State;
    /* +0x066 */ unsigned char            ThreadSeed;
    /* +0x067 */ unsigned char            DisableBoost;
    /* +0x068 */ unsigned char            PowerState;
    /* +0x069 */ unsigned char            DisableQuantum;
    /* +0x06a */ unsigned char            IdealNode;
    /* +0x06b  KEXECUTE_OPTIONS Flags; */
    /* +0x06b */ unsigned char            ExecuteOptions;
};

struct EPROCESS {
    /* +0x000 */ struct KPROCESS      Pcb;
    /* +0x06c */ unsigned long        ProcessLock;
    /* +0x070 */ unsigned long long   CreateTime;
    /* +0x078 */ unsigned long long   ExitTime;
    /* +0x080 */ unsigned long        RunDownProtect;
    /* +0x084 */ void                *UniqueProcessId;
    /* +0x088 */ struct LIST_ENTRY    ActiveProcessLinks;
    /* +0x090 */ unsigned long        QuotaUsage[3];
    /* +0x09c */ unsigned long        QuotaPeak[3];
    /* +0x0a8 */ unsigned long        CommitCharge;
    /* +0x0ac */ unsigned long        PeakVirtualSize;
    /* +0x0b0 */ unsigned long        VirtualSize;
    /* +0x0b4 */ struct LIST_ENTRY    SessionProcessLinks;
    /* +0x0bc */ void                *DebugPort;
    /* +0x0c0 */ void                *ExceptionPort;
    /* +0x0c4 */ struct HANDLE_TABLE *ObjectTable;
    /* +0x0c8 */ unsigned long        Token;
    /* +0x0cc */ struct FAST_MUTEX    WorkingSetLock;
    /* +0x0ec */ unsigned long        WorkingSetPage;
    /* +0x0f0 */ struct FAST_MUTEX    AddressCreationLock;
    /* +0x110 */ unsigned long        HyperSpaceLock;
    /* +0x114 */ struct ETHREAD      *ForkInProgress;
    /* +0x118 */ unsigned long        HardwareTrigger;
    /* +0x11c */ void                *VadRoot;
    /* +0x120 */ void                *VadHint;
    /* +0x124 */ void                *CloneRoot;
    /* +0x128 */ unsigned long        NumberOfPrivatePages;
    /* +0x12c */ unsigned long        NumberOfLockedPages;
    /* +0x130 */ void                *Win32Process;
    /* +0x134 */ void                *Job; /*Points to EJOB*/
    /* +0x138 */ void                *SectionObject;
    /* +0x13c */ void                *SectionBaseAddress;
    /* +0x140 */ void   *QuotaBlock; /*points to EPROCESS_QUOTA_BLOCK*/
    /* +0x144 */ void  *WorkingSetWatch;/*points to PAGEFAULT_HISTORY*/
    /* +0x148 */ void                *Win32WindowStation;
    /* +0x14c */ void                *InheritedFromUniqueProcessId;
    /* +0x150 */ void                *LdtInformation;
    /* +0x154 */ void                *VadFreeHint;
    /* +0x158 */ void                *VdmObjects;
    /* +0x15c */ struct DEVICE_MAP   *DeviceMap;
    /* +0x160 */ struct LIST_ENTRY    PhysicalVadList;
    /* +0x168 */ struct HARDWARE_PTE  PageDirectoryPte;
    /* +0x16c */ unsigned long        Filler;
    /* +0x170 */ void                *Session;
    /* +0x174 */ char                 ImageFileName[16];
    /* +0x184 */ struct LIST_ENTRY    JobLinks;
    /* +0x18c */ void                *LockedPagesList;
    /* +0x190 */ struct LIST_ENTRY    ThreadListHead;
    /* +0x198 */ void                *SecurityPort;
    /* +0x19c */ void                *PaeTop;
    /* +0x1a0 */ unsigned long        ActiveThreads;
    /* +0x1a4 */ unsigned long        GrantedAccess;
    /* +0x1a8 */ unsigned long        DefaultHardErrorProcessing;
    /* +0x1ac */ long                 LastThreadExitStatus;
    /* +0x1b0 */ struct PEB          *Peb;
    /* +0x1b4 */ unsigned long        PrefetchTrace;
    /* +0x1b8 */ unsigned long long   ReadOperationCount;
    /* +0x1c0 */ unsigned long long   WriteOperationCount;
    /* +0x1c8 */ unsigned long long   OtherOperationCount;
    /* +0x1d0 */ unsigned long long   ReadTransferCount;
    /* +0x1d8 */ unsigned long long   WriteTransferCount;
    /* +0x1e0 */ unsigned long long   OtherTransferCount;
```

```
    /* +0x1e8 */ unsigned long        CommitChargeLimit;
    /* +0x1ec */ unsigned long        CommitChargePeak;
    /* +0x1f0 */ void                *AweInfo;
    /* +0x1f4 */ unsigned long        SeAuditProcessCreationInfo;
    /* +0x1f8 */ struct MMSUPPORT     Vm;
    /* +0x238 */ unsigned long        LastFaultCount;
    /* +0x23c */ unsigned long        ModifiedPageCount;
    /* +0x240 */ unsigned long        NumberOfVads;
    /* +0x244 */ unsigned long        JobStatus;
    /* +0x248 */ unsigned long        Flags;
    /* +0x24c */ long                 ExitStatus;
    /* +0x250 */ unsigned short       NextPageColor;
    /* +0x252 */ unsigned char        SubSystemMinorVersion;
    /* +0x253 */ unsigned char        SubSystemMajorVersion;
    /* +0x254 */ unsigned short       SubSystemVersion;
    /* +0x256 */ unsigned char        PriorityClass;
    /* +0x257 */ unsigned char        WorkingSetAcquiredUnsafe;
    /* +0x258 */ unsigned long        Cookie;
};

struct CLIENT_ID {
    /* +0x000 */ void *UniqueProcess;
    /* +0x004 */ void *UniqueThread;
};

struct KAPC_STATE {
    /* +0x000 */ struct LIST_ENTRY  ApcListHead[2];
    /* +0x010 */ struct KPROCESS   *Process;
    /* +0x014 */ unsigned char      KernelApcInProgress;
    /* +0x015 */ unsigned char      KernelApcPending;
    /* +0x016 */ unsigned char      UserApcPending;
      /* +0x017 */ unsigned char      Trailer;
};

struct KQUEUE {
    /* +0x000 */ struct DISPATCHER_HEADER Header;
    /* +0x010 */ struct LIST_ENTRY        EntryListHead;
    /* +0x018 */ unsigned long            CurrentCount;
    /* +0x01c */ unsigned long            MaximumCount;
    /* +0x020 */ struct LIST_ENTRY        ThreadListHead;
};

struct KTIMER {
    /* +0x000 */ struct DISPATCHER_HEADER Header;
    /* +0x010 */ unsigned long long       DueTime;
    /* +0x018 */ struct LIST_ENTRY        TimerListEntry;
    /* +0x020 */ struct KDPC             *Dpc;
    /* +0x024 */ long                     Period;
};

struct KAPC {
    /* +0x000 */ short                    Type;
    /* +0x002 */ short                    Size;
    /* +0x004 */ unsigned long            Spare0;
    /* +0x008 */ struct KTHREAD          *Thread;
    /* +0x00c */ struct LIST_ENTRY        ApcListEntry;
    /* +0x014 */ void                    *KernelRoutine;
    /* +0x018 */ void                    *RundownRoutine;
    /* +0x01c */ void                    *NormalRoutine;
    /* +0x020 */ void                    *NormalContext;
    /* +0x024 */ void                    *SystemArgument1;
    /* +0x028 */ void                    *SystemArgument2;
    /* +0x02c */ char                     ApcStateIndex;
    /* +0x02d */ char                     ApcMode;
    /* +0x02e */ unsigned char            Inserted;
      /* +0x02f */ char                     Trailer;
};

struct KWAIT_BLOCK {
    /* +0x000 */ struct LIST_ENTRY   WaitListEntry;
    /* +0x008 */ struct KTHREAD     *Thread;
    /* +0x00c */ void               *Object;
    /* +0x010 */ struct KWAIT_BLOCK *NextWaitBlock;
    /* +0x014 */ unsigned short      WaitKey;
    /* +0x016 */ unsigned short      WaitType;
};

struct KTHREAD {
    /* +0x000 */ struct DISPATCHER_HEADER  Header;
    /* +0x010 */ struct LIST_ENTRY         MutantListHead;
    /* +0x018 */ void                     *InitialStack;
    /* +0x01c */ void                     *StackLimit;
    /* +0x020 */ void                     *Teb;
    /* +0x024 */ void                     *TlsArray;
    /* +0x028 */ void                     *KernelStack;
    /* +0x02c */ unsigned char             DebugActive;
    /* +0x02d */ unsigned char             State;
    /* +0x02e */ unsigned char             Alerted[2];
    /* +0x030 */ unsigned char             Iopl;
    /* +0x031 */ unsigned char             NpxState;
    /* +0x032 */ char                      Saturation;
    /* +0x033 */ char                      Priority;
    /* +0x034 */ struct KAPC_STATE         ApcState;
    /* +0x04c */ unsigned long             ContextSwitches;
    /* +0x050 */ unsigned char             IdleSwapBlock;
    /* +0x051 */ unsigned char             Spare0[3];
    /* +0x054 */ long                      WaitStatus;
    /* +0x058 */ unsigned char             WaitIrql;
    /* +0x059 */ char                      WaitMode;
    /* +0x05a */ unsigned char             WaitNext;
    /* +0x05b */ unsigned char             WaitReason;
    /* +0x05c */ struct KWAIT_BLOCK       *WaitBlockList;
    /* +0x060 */ struct LIST_ENTRY         WaitListEntry;
    /* +0x068 */ unsigned long             WaitTime;
    /* +0x06c */ char                      BasePriority;
    /* +0x06d */ unsigned char             DecrementCount;
    /* +0x06e */ char                      PriorityDecrement;
    /* +0x06f */ char                      Quantum;
    /* +0x070 */ struct KWAIT_BLOCK        WaitBlock[4];
    /* +0x0d0 */ void                     *LegoData;
    /* +0x0d4 */ unsigned long             KernelApcDisable;
    /* +0x0d8 */ unsigned long             UserAffinity;
    /* +0x0dc */ unsigned char             SystemAffinityActive;
    /* +0x0dd */ unsigned char             PowerState;
    /* +0x0de */ unsigned char             NpxIrql;
    /* +0x0df */ unsigned char             InitialNode;
    /* +0x0e0 */ void                     *ServiceTable;
    /* +0x0e4 */ struct KQUEUE            *Queue;
    /* +0x0e8 */ unsigned long             ApcQueueLock;
    /* +0x0f0 */ struct KTIMER             Timer;
    /* +0x118 */ struct LIST_ENTRY         QueueListEntry;
    /* +0x120 */ unsigned long             SoftAffinity;
    /* +0x124 */ unsigned long             Affinity;
    /* +0x128 */ unsigned char             Preempted;
    /* +0x129 */ unsigned char             ProcessReadyQueue;
    /* +0x12a */ unsigned char             KernelStackResident;
    /* +0x12b */ unsigned char             NextProcessor;
    /* +0x12c */ void                     *CallbackStack;
    /* +0x130 */ void                     *Win32Thread;
    /* +0x134 */ void              *TrapFrame;/*points to KTRAP_FRAME*/
    /* +0x138 */ struct KAPC_STATE        *ApcStatePointer[2];
    /* +0x140 */ char                      PreviousMode;
    /* +0x141 */ unsigned char             EnableStackSwap;
    /* +0x142 */ unsigned char             LargeStack;
    /* +0x143 */ unsigned char             ResourceIndex;
    /* +0x144 */ unsigned long             KernelTime;
    /* +0x148 */ unsigned long             UserTime;
    /* +0x14c */ struct KAPC_STATE         SavedApcState;
    /* +0x164 */ unsigned char             Alertable;
    /* +0x165 */ unsigned char             ApcStateIndex;
    /* +0x166 */ unsigned char             ApcQueueable;
    /* +0x167 */ unsigned char             AutoAlignment;
    /* +0x168 */ void                     *StackBase;
    /* +0x16c */ struct KAPC               SuspendApc;
    /* +0x19c */ struct KSEMAPHORE         SuspendSemaphore;
    /* +0x1b0 */ struct LIST_ENTRY         ThreadListEntry;
    /* +0x1b8 */ char                      FreezeCount;
    /* +0x1b9 */ char                      SuspendCount;
    /* +0x1ba */ unsigned char             IdealProcessor;
    /* +0x1bb */ unsigned char             DisableBoost;
    /* +0x1bc */ long long                 Trailer;
};

struct ETHREAD {
    /* +0x000 */ struct KTHREAD       Tcb;
    /* +0x1c0 */ long long            CreateTime;
    /* +0x1c8 */ long long            ExitTime;
    /* +0x1d0 */ long                 ExitStatus;
    /* +0x1d4 */ struct LIST_ENTRY    PostBlockList;
    /* +0x1dc */ void                *TerminationPort;
    /* +0x1e0 */ unsigned long        ActiveTimerListLock;
    /* +0x1e4 */ struct LIST_ENTRY    ActiveTimerListHead;
    /* +0x1ec */ struct CLIENT_ID     Cid;
    /* +0x1f4 */ struct KSEMAPHORE    LpcReplySemaphore;
    /* +0x208 */ void                *LpcReplyMessage;
    /* +0x20c */ void                *ImpersonationInfo;
    /* +0x210 */ struct LIST_ENTRY    IrpList;
    /* +0x218 */ unsigned long        TopLevelIrp;
    /* +0x21c */ void                *DeviceToVerify;
    /* +0x220 */ struct EPROCESS     *ThreadsProcess;
    /* +0x224 */ void                *StartAddress;
    /* +0x228 */ void                *Win32StartAddress;
    /* +0x22c */ struct LIST_ENTRY    ThreadListEntry;
    /* +0x234 */ unsigned long        RundownProtect;
    /* +0x238 */ unsigned long        ThreadLock;
    /* +0x23c */ unsigned long        LpcReplyMessageId;
    /* +0x240 */ unsigned long        ReadClusterSize;
    /* +0x244 */ unsigned long        GrantedAccess;
    /* +0x248 */ unsigned long        CrossThreadFlags;
    /* +0x24c */ unsigned long        SameThreadPasssiveFlags;
    /* +0x250 */ unsigned long        SameThreadApcFlags;
    /* +0x254 */ unsigned char        ForwardClusterOnly;
    /* +0x255 */ unsigned char        DisablePageFaultClustering;
};

__END_DECLS

#endif
```

APPENDIX B. SOURCE CODE

A. EPROC.C

```
/**********************************************************************
File eprocess.c
Version 1.0
Author Jared Stimson

Description: This is a tool used to search for processes using various
methods.  It can be used to gather information from the image about
running processes.
**********************************************************************/
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
#include<time.h>
#include<io.h>
#include<windows.h>
#include<ctype.h>
#include<unistd.h>
#include"win32structs.h"

#define KERNEL_OFFSET               0x80000000
#define SIZEOF_PROC                 0x260
#define SIZEOF_THRD                 0x258
#define PAGESIZE              0x1000
#define TOTAL_TYPES                 32

int PAGEFILE = 0;

/**********************************************************************
Prints the usage statement.
Input: None
Output: Usage Statement
**********************************************************************/
void print_usage(){
      fprintf (stderr, "Usage: ./eprocess2DEC07.exe option <imageFile>
<args>\n"
            " -l Walk active process list\n"
            " -s Scan image for process structures\n"
            " -t Translates Virtual to Physical Address\n"
            " -d Dump process information\n"
            " -i Identify Operating System\n"
            " -p Gets the number of pages in memory\n"
            " -m Dump Process Memory\n");
}

/**********************************************************************
Convert the hex string to int.
Input:  Hex String
Output: int
**********************************************************************/
```

```
unsigned int hstr_i(char *cptr){
      unsigned int i, j = 0;
      while (cptr && *cptr && isxdigit(*cptr)){
            i = *cptr++ - '0';
            if (9 < i)
                  i -= 7;
            j <<= 4;
            j |= (i & 0x0f);
      }
      return(j);
}

/*********************************************************************
Convert the Windows time format to Unix format and convert the Unix
time
to GMT with the existing Linux Time library functions.

       Unix    epoch is 1970-01-01 00:00:00 resolution is seconds
       Windows epoch is 1601-01-01 00:00:00 resolution is 100ns
*********************************************************************/
void win_time(long long winTime, char date[]){
      long unixTime = 0;
      unixTime = (long long) (winTime/10000000) - (long long)
11644473600ULL;
            strftime(date,32,"%Y-%m-%d %H:%M:%S %Z",gmtime(&unixTime));
}

/*********************************************************************
Translates virtual address to a valid physical address, if the virtual
address is mapped to a physical address.
Input: Long, Page Directory Base, File descriptor, Size of file.
Output: Pyisical offset of virtual address (long)
*********************************************************************/
long virtual_to_physical(long ptr32, long pageDirBase, FILE * in,long
size) {
      if(size > 256000000){
            if((ptr32 & 0xf0000000) <= 0x80000000 )
                  return ptr32 - KERNEL_OFFSET;
      }
      unsigned long pageDirectoryIndex = ptr32 & 0xffc00000;
      pageDirectoryIndex = pageDirectoryIndex >> 22;
      unsigned long pageTableIndex = ptr32 & 0x003ff000;
      pageTableIndex = pageTableIndex >> 12;
      unsigned long byteIndex = ptr32 & 0x00000fff;
      ptr32 = pageDirBase + pageDirectoryIndex * 4;
      fseek(in,ptr32,SEEK_SET);
      fread(&ptr32,4,1,in);
      if( (ptr32 & 0x01) || ((ptr32 & 0xc00) == 0) || ((ptr32 & 0xc00)
== 0x8)) {
            ptr32 = (ptr32 & 0xfffff000) + pageTableIndex * 4;
            if(ptr32 == 0) return -1;
            fseek(in,ptr32,SEEK_SET);
            fread(&ptr32,4,1,in);
            if((ptr32 & 0x01) || ((ptr32 & 0xc00) == 0) || ((ptr32 &
0xc00) == 0x8)) {
                  ptr32 = (ptr32 & 0xfffff000) + byteIndex;
                  return ptr32;
            } else return -1;
      } else return -1;
}

/**********************************************************************
Returns the page Table Base.
Input: Pointer to Page Table Base
Output: Contents of PTB
**********************************************************************/
long get_PTB(long ptr32) {
      long pageTableBase = 0;
      if((ptr32 & 1)){
            pageTableBase = ptr32 & 0xfffff000;
            return pageTableBase;
      }
      else {
            if(!(ptr32 & 0xc00)){
                  if((ptr32 & 0x1e) == 0){
                        pageTableBase = (ptr32 & 0xfffff000);
                        if(!pageTableBase)
                              return 0;
                        pageTableBase = pageTableBase + 1;
                        return pageTableBase;
                  } else
                        return 3;
            }
            if(ptr32 & 0x400)
                  pageTableBase = pageTableBase + 2;
      }
      return 4;
}

/**********************************************************************
Returns the page Base Address.
Input: Pointer to Page Base Address
Output: Contents of Page Base Address
**********************************************************************/
long get_PBA(long ptr32) {
      long pageBaseAddr = 0;
      if(ptr32 & 0x100){
            return 5;
      }
      if((ptr32 & 1) || ((ptr32 & 0xc00) == 0x800)){
            pageBaseAddr = ptr32 & 0xfffff000;
            return pageBaseAddr;
      } else {
            if((ptr32 & 0xc00) == 0){
                  if((ptr32 & 0x1e) == 0){
                        pageBaseAddr = ptr32 & 0xfffff000;
                        if(!pageBaseAddr)
                              return 0;
                        pageBaseAddr = pageBaseAddr + 1;
                        return pageBaseAddr;
                  } else
                        return 3;
            }
            if(ptr32 & 0x400)
```

```
                  return 2;
      }
      return 4;
}

/**********************************************************************
Convert unicode string to ascii string and put it in tempString.
Input: unicode string
Output: ascii string
**********************************************************************/
void unicode_to_ascii(char *string, unsigned short length) {
      if( length <= 0 )
            return;
      int m;
      char *tempString = malloc(length/2);
      for(m=0;m<length;m+=2) {
            memcpy(tempString+m/2,string+m,1);
      }
      memset(string,0,length);
      memcpy(string,tempString,length/2);
      free(tempString);
      return;
}

/**********************************************************************
Compares 2 4-byte numbers for greater than, less than, or equal to
condition.
Input: 2 numbers
Output: -1,0,1
**********************************************************************/
int longcmp( const void *n1, const void *n2 ) {
      unsigned long a = *(unsigned long *)n1;
      unsigned long b = *(unsigned long *)n2;

      return (a < b) ? -1 : ((a == b) ? 0 : 1);
}

/**********************************************************************
Used to lookup the machine type for an image.
Input: machine number, empty string
Output: machine name
**********************************************************************/
void lookup_machine(int machineNumber, char machineName[]){
      switch (machineNumber) {
            case 0x014c:
                  strcpy(machineName,"IMAGE_FILE_MACHINE_I386");
                  break;
            case 0x014d:
                  strcpy(machineName,"IMAGE_FILE_MACHINE_I860");
                  break;
            case 0x0200:
                  strcpy(machineName,"IMAGE_FILE_MACHINE_IA64");
                  break;
            case 0x8664:
                  strcpy(machineName,"IMAGE_FILE_MACHINE_AMD64");
                  break;
            default:
                  strcpy(machineName,"Machine Unknown");
                  break;
      }
}

/***********************************************************************
Returns String with file header characteristic information.
Input: Empty string
Output: Characteristics of file image
***********************************************************************/
void get_file_header_characteristics(int charsIn, char
characteristics[]){
      if(charsIn & 0x0001)
            strcat(characteristics, "\tIMAGE_FILE_RELOCS_STRIPPED\n");
      if(charsIn & 0x0002)
            strcat(characteristics, "\tIMAGE_FILE_EXECUTABLE_IMAGE\n");
      if(charsIn & 0x0004)
            strcat(characteristics,
"\tIMAGE_FILE_LINE_NUMS_STRIPPED\n");
      if(charsIn & 0x0008)
            strcat(characteristics,
"\tIMAGE_FILE_LOCAL_SYMS_STRIPPED\n");
      if(charsIn & 0x0010)
            strcat(characteristics,
"\tIMAGE_FILE_AGGRESIVE_WS_TRIM\n");
      if(charsIn & 0x0020)
            strcat(characteristics,
"\tIMAGE_FILE_LARGE_ADDRESS_AWARE\n");
      if(charsIn & 0x0080)
            strcat(characteristics,
"\tIMAGE_FILE_BYTES_REVERSED_LO\n");
      if(charsIn & 0x0100)
            strcat(characteristics, "\tIMAGE_FILE_32BIT_MACHINE\n");
      if(charsIn & 0x0200)
            strcat(characteristics, "\tIMAGE_FILE_DEBUG_STRIPPED\n");
      if(charsIn & 0x0400)
            strcat(characteristics,
"\tIMAGE_FILE_REMOVABLE_RUN_FROM_SWAP\n");
      if(charsIn & 0x0800)
            strcat(characteristics,
"\tIMAGE_FILE_NET_RUN_FROM_SWAP\n");
      if(charsIn & 0x1000)
            strcat(characteristics, "\tIMAGE_FILE_SYSTEM\n");
      if(charsIn & 0x2000)
            strcat(characteristics, "\tIMAGE_FILE_DLL\n");
      if(charsIn & 0x4000)
            strcat(characteristics, "\tIMAGE_FILE_UP_SYSTEM_ONLY\n");
      if(charsIn & 0x8000)
            strcat(characteristics,
"\tIMAGE_FILE_BYTES_REVERSED_HI\n");
      strcat(characteristics,"");
}

/***********************************************************************
Returns String with Optional Header Subsystem Information.
Input: Empty string
Output: File subsystem information
```

```
**********************************************************************/
void get_optional_header_subsystem(int subIn, char subSystem[]){
      switch (subIn) {
            case 0:
                  strcpy(subSystem,"IMAGE_SUBSYSTEM_UNKNOWN");
                  break;
            case 1:
                  strcpy(subSystem,"IMAGE_SUBSYSTEM_NATIVE");
                  break;
            case 2:
                  strcpy(subSystem,"IMAGE_SUBSYSTEM_WINDOWS_GUI");
                  break;
            case 3:
                  strcpy(subSystem,"IMAGE_SUBSYSTEM_WINDOWS_CUI");
                  break;
            case 5:
                  strcpy(subSystem,"IMAGE_SUBSYSTEM_OS2_CUI");
                  break;
            case 7:
                  strcpy(subSystem,"IMAGE_SUBSYSTEM_POSIX_CUI");
                  break;
            case 8:
                  strcpy(subSystem,"IMAGE_SUBSYSTEM_NATIVE_WINDOWS");
                  break;
            case 9:
                  strcpy(subSystem,"IMAGE_SUBSYSTEM_WINDOWS_CE_GUI");
                  break;
            case 14:
                  strcpy(subSystem,"IMAGE_SUBSYSTEM_XBOX");
                  break;
            default:
                  strcpy(subSystem,"Machine Unknown");
                  break;
      }

}

/**********************************************************************
Prints Data Directory Information
Input: Image Optional Header
Output: Data Directory Information

Directory Entries
#define IMAGE_DIRECTORY_ENTRY_EXPORT         0
#define IMAGE_DIRECTORY_ENTRY_IMPORT         1
#define IMAGE_DIRECTORY_ENTRY_RESOURCE       2
#define IMAGE_DIRECTORY_ENTRY_EXCEPTION      3
#define IMAGE_DIRECTORY_ENTRY_SECURITY       4
#define IMAGE_DIRECTORY_ENTRY_BASERELOC      5
#define IMAGE_DIRECTORY_ENTRY_DEBUG          6
#define IMAGE_DIRECTORY_ENTRY_COPYRIGHT      7
#define IMAGE_DIRECTORY_ENTRY_GLOBALPTR      8
#define IMAGE_DIRECTORY_ENTRY_TLS            9
#define IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG    10

***********************************************************************
/
```

```
void print_data_directories(struct _IMAGE_OPTIONAL_HEADER *optHeader){
            printf("\n\t\tData Directory Information\n");
            printf("\n%-27s     %-10s     %-10s\n", "Data Directory",
"RVA", "Size");
            printf("%-27s     %-10s     %-10s\n", "--------------", "--
-", "----");
            char directory[11][32] = {"Export Directory", "Import
Directory", "Resource Directory",
                                    "Exception Directory","Security
Directory", "Base Relocation Table",
                                    "Debug Directory", "Description
String", "Machine Value (MIPS GP)",
                                    "TLS Directory", "Load
Configuration Directory"};
            int i = 0;
            for(i;i < 11; i++){
                  if((optHeader->DataDirectory[i].VirtualAddress  > 0)
&& (optHeader->DataDirectory[i].Size > 0))
                        printf("%-32s0x%.8x     0x%.8x\n",
                              directory[i],
                              optHeader-
>DataDirectory[i].VirtualAddress,
                              optHeader->DataDirectory[i].Size);
            }
}

/**********************************************************************
Prints section header information.
Input: File, offset, Image section header
Output: Section information
**********************************************************************/
void print_section_data(FILE *in, int offset, struct
_IMAGE_SECTION_HEADER *sectHeader){
      fseek(in,offset,SEEK_SET);
      fread(sectHeader,sizeof(struct _IMAGE_SECTION_HEADER),1,in);
      memcpy(sectHeader->Name + 8,"",1);
      printf("%-9s0x%.8x  0x%.8x  0x%.8x  0x%.8x  0x%.8x\n",
            sectHeader->Name,
            sectHeader->Misc.VirtualSize,
            sectHeader->VirtualAddress,
            sectHeader->SizeOfRawData,
            sectHeader->PointerToRawData,
            sectHeader->Characteristics);
}

/**********************************************************************
Calculates statistics about memory usage.
Input: Eprocess block, File, page file, size
Output: Memory Stats
**********************************************************************/
void get_memory_stats(struct EPROCESS *eprocess, FILE *mem, FILE
*pageFile, int size){
      long PDE[1024];
      long PTE[1024];
      long pageTableBase = 0;
      int i = 0;
      int valid = 0;
      int paged = 0;
      int zero = 0;
      int global = 0;
      int prototype = 0;
      int unknown = 0;
      fseek(mem,eprocess->Pcb.DirectoryTableBase[0],SEEK_SET);
      fread(PDE,4096,1,mem);
      for(i;i < 1024;i++){
            pageTableBase = get_PTB(PDE[i]);
            if(pageTableBase != 0) {
                  switch (pageTableBase & 0xf) {
                        case 0:
                              fseek(mem,pageTableBase,SEEK_SET);
                              fread(PTE,4096,1,mem);
                              break;
                        case 1:
                              if(pageFile != NULL) {

      fseek(pageFile,pageTableBase,SEEK_SET);
                                    fread(PTE,4096,1,pageFile);
                              } else
                                    pageTableBase = 0;
                              break;
                        default:
                              pageTableBase = 0;
                              break;
                  }
                  if(pageTableBase != 0) {
                        int j = 0;
                        long pageBaseAddr = 0;
                        for(j;j<1024;j++){
                              pageBaseAddr = get_PBA(PTE[j]);
                              if(pageBaseAddr != 0) {
                                    switch (pageBaseAddr & 0xf) {
                                          case 0:
                                                valid++;
                                                break;
                                          case 1:
                                                paged++;
                                                break;
                                          case 2:
                                                prototype++;
                                                break;
                                          case 3:
                                                unknown++;
                                                break;
                                          case 5:
                                                global++;
                                                break;
                                    }
                              }
                        }
                  }
            }
      }
      printf("\nValid\t\t%d\nPrototype\t%d\nGlobal\t\t%d\nPaged\t\t%d\n
Unknown\t\t%d\n",
            valid, prototype, global, paged, unknown);
}
/**********************************************************************
Dumps the processes memory pages from the image and optionaly the
paging
file if available using the Page Directories.
**********************************************************************/
void get_memory(struct EPROCESS *eprocess, FILE *mem, FILE *pageFile,
int size){
      char pname[24];
      char processID[8];
      char fname[56];
      char iname[56];
      char directory[24] = "mkdir ";
      char deldir[24] = "rm -rf ";
      byte buffer[4096];
      long pid = 0;
      long PDE[1024];
      long PTE[1024];
      long pageTableBase = 0;
      char vAddr[20];
      int i = 0;
      int valid = 0;
      int paged = 0;
      int zero = 0;
      int global = 0;
      int prototype = 0;
      int unknown = 0;
      FILE *fd1 = NULL;
      FILE *fd2 = NULL;
      fseek(mem,eprocess->Pcb.DirectoryTableBase[0],SEEK_SET);
      fread(PDE,4096,1,mem);
      pid = (long)eprocess->UniqueProcessId;
      snprintf(processID,sizeof(processID),"\x5f%lu",pid);
      strncpy(pname,eprocess->ImageFileName,sizeof(pname));
      strncat(pname,processID,sizeof(pname));
      strncat(directory,pname,sizeof(directory)-strlen(directory));
      if(fopen(pname,"r") != NULL ) {
            fprintf(stderr,"Error: Directory %s already
exists!\n",pname);
            exit(0);
      }
      system(directory);
      strncpy(fname,pname,sizeof(fname));
            strncat(fname,"/",1);
      strncat(fname,pname,sizeof(fname)-strlen(fname));
      strncpy(iname,fname,sizeof(fname));
      strncat(fname,".dmp",4);
      strncat(iname,".txt",4);
      fd1 = fopen(fname,"wb");
      fd2 = fopen(iname,"wb");
      if( fd1 == NULL ){
                  fprintf(stderr,"Error: Could not open %s!\n",fname);
            exit(0);
      }
      if( fd2 == NULL ){
                  fprintf(stderr,"Error: Could not open %s!\n",iname);
            exit(0);
      }
      fprintf(stderr,"Creating Working Set Files, Please Wait\n");
      for(i;i < 1024;i++){
            pageTableBase = get_PTB(PDE[i]);
            if(pageTableBase != 0) {
                  switch (pageTableBase & 0xf) {
                        case 0:
                              fseek(mem,pageTableBase,SEEK_SET);
                              fread(PTE,4096,1,mem);
                              break;
                        case 1:
                              if(pageFile != NULL) {

      fseek(pageFile,pageTableBase,SEEK_SET);
                                    fread(PTE,4096,1,pageFile);
                              } else
                                    pageTableBase = 0;
                              break;
                        default:
                              pageTableBase = 0;
                              break;
                  }
                  if(pageTableBase != 0) {
                        int j = 0;
                        long pageBaseAddr = 0;
                        for(j;j<1024;j++){
                              pageBaseAddr = get_PBA(PTE[j]);
                              if(pageBaseAddr != 0) {
                                    sprintf(vAddr,"%8x",((i << 22) + (j
<< 12)));
                                    switch (pageBaseAddr & 0xf) {
                                          case 0:

      fseek(mem,pageBaseAddr,SEEK_SET);

      fread(buffer,1,sizeof(buffer),mem);

      fwrite(buffer,1,sizeof(buffer),fd1);

      fprintf(fd2,"%s\n",vAddr);
                                                valid++;
                                                break;
                                          case 1:
                                                if(pageFile != NULL){

      fseek(pageFile,pageBaseAddr,SEEK_SET);

      fread(buffer,1,sizeof(buffer),pageFile);

      fwrite(buffer,1,sizeof(buffer),fd1);

      fprintf(fd2,"%s\n",vAddr);
                                                }
                                                paged++;
                                                break;
                                          case 2:
                                                prototype++;
                                                break;
                                          case 3:
                                                unknown++;
                                                break;
                                          case 5:
                                                global++;
                                                break;
                                    }
                              }
                        }
                  }
            }
      }
      fclose(fd1);
      fclose(fd2);
      printf("\nValid\t\t%d\nPrototype\t%d\nGlobal\t\t%d\nPaged\t\t%d\n
Unknown\t\t%d\n",
            valid, prototype, global, paged, unknown);
}
/**********************************************************************
Prints Optional Header information from process image.
Input: Image ofptional header
Output: Prints optional header information
**********************************************************************/
void print_optional_header(struct _IMAGE_OPTIONAL_HEADER *optHeader){
            char subSystem[] = "Subsystem Unknown";
            get_optional_header_subsystem(optHeader->Subsystem,
subSystem);
            printf("\n\t\tOptional Header\n");
            printf("Magic Number:                   0x%x\n", optHeader-
>Magic);
            printf("Subsystem:                      %s\n", subSystem);
            printf("Image Base Address:             0x%x\n", optHeader-
>ImageBase);
            printf("Address of Entry Point:         0x%x (RVA)\n",
optHeader->AddressOfEntryPoint);
            printf("Code Base Address:              0x%x (RVA)\n",
optHeader->BaseOfCode);
            printf("Data Base Address:              0x%x (RVA)\n",
optHeader->BaseOfData);
            printf("Major OS Version:               %d\n", optHeader-
>MajorOperatingSystemVersion);
            printf("Minor OS Version:               %d\n", optHeader-
>MinorOperatingSystemVersion);
            printf("Size of Stack Reserved:         0x%x (%d bytes)\n",
optHeader->SizeOfStackReserve, optHeader->SizeOfStackReserve);
            printf("Number of RVA and Sizes:        0x%x\n", optHeader-
>NumberOfRvaAndSizes);
}

/**********************************************************************
Prints DOS Header informaiton from process image.
Input: File, Image Dos header
Output: Dos header information
**********************************************************************/
```

```
void print_dos_header(FILE * in,struct _IMAGE_DOS_HEADER
*dosHeader,long imgBaseOffset){
      printf("\n\t\tDOS Header\n");
      printf("e_magic:                        0x%x\n",dosHeader-
>e_magic);
      printf("e_lfanew:                       0x%x\n",dosHeader-
>e_lfanew);
      int ntHeader = 0;
      fseek(in,(long) imgBaseOffset + dosHeader->e_lfanew, SEEK_SET);
      fread(&ntHeader,2,1,in);
      printf("SIGNATURE:                      0x%x\n",ntHeader);
}

/**********************************************************************
Prints File Header informaiton.
Input: Image File Header
Output: Prints file header information
**********************************************************************/
void print_file_header(struct _IMAGE_FILE_HEADER *fileHeader){
      char machineName[] = "Machine Unknown";
      char timeStamp[32];
      char characteristics[256];
      strcpy(characteristics,"\n");
      lookup_machine(fileHeader->Machine,machineName);
      printf("\n\t\tFile Header\n");
      printf("Machine Type:                   %s\n", machineName);
      printf("Number of Sections:             %d\n", fileHeader-
>NumberOfSections);
      win_time(fileHeader->TimeDateStamp, timeStamp);
      printf("Time Stamp:                     %s\n", timeStamp);
      printf("Size of Optional Header:        0x%x (%d bytes)\n",
fileHeader->SizeOfOptionalHeader, fileHeader->SizeOfOptionalHeader);
      get_file_header_characteristics(fileHeader-
>Characteristics,characteristics);
      printf("Characteristics: \n%s\n", characteristics);
}

/**********************************************************************
Prints the process image information.
Input: Eprocess block, file, size, binary verbose
Output: kernel image information
**********************************************************************/
void print_image_info(struct EPROCESS *eprocess, FILE * in, long size,
int b){
      struct PEB *peb;
      int peb_offset = 0;
      int imgBaseOffset = 0;
      int address = 0;
      byte buf[1000];
      peb = malloc(sizeof(struct PEB));
      peb_offset = virtual_to_physical((long) eprocess->Peb,eprocess-
>Pcb.DirectoryTableBase[0],in, size);
      fseek(in,(long) peb_offset,SEEK_SET);
      fread(peb,sizeof(struct PEB),1,in);
      imgBaseOffset = virtual_to_physical((long) peb-
>ImageBaseAddress,eprocess->Pcb.DirectoryTableBase[0],in, size);
      if(imgBaseOffset != -1){
            printf("\nImgage Base Offset: 
0x%x\n",imgBaseOffset);
            struct _IMAGE_DOS_HEADER *dosHeader;
            dosHeader = malloc(sizeof(struct _IMAGE_DOS_HEADER));
            fseek(in,(long) imgBaseOffset, SEEK_SET);
            fread(dosHeader,64,1,in);
            print_dos_header(in,dosHeader,imgBaseOffset);
            struct _IMAGE_FILE_HEADER *fileHeader;
            fileHeader = malloc(sizeof(struct _IMAGE_FILE_HEADER));
            fseek(in,(long) imgBaseOffset + dosHeader->e_lfanew + 4, 
SEEK_SET);
            fread(fileHeader,sizeof(struct _IMAGE_FILE_HEADER),1,in);
            print_file_header(fileHeader);
            struct _IMAGE_OPTIONAL_HEADER *optHeader;
            optHeader = malloc(sizeof(struct _IMAGE_OPTIONAL_HEADER));
            fseek(in,imgBaseOffset + dosHeader->e_lfanew + 
24,SEEK_SET);
            fread(optHeader,sizeof(struct 
_IMAGE_OPTIONAL_HEADER),1,in);
            print_optional_header(optHeader);
            print_data_directories(optHeader);
            printf("\n\t\tSection Information\n");
            struct _IMAGE_SECTION_HEADER *sectHeader[fileHeader-
>NumberOfSections];
            printf("\nName     Virt Size   Virt Addr   rData Ofs   
rData Size  Char\n");
            printf("----     ---------   ---------   ---------   ------
----  ----\n");
            int i = 0;
            for(i;i < fileHeader->NumberOfSections;i++){
                  sectHeader[i] = malloc(sizeof(struct 
_IMAGE_SECTION_HEADER));
                  print_section_data(in, imgBaseOffset + dosHeader-
>e_lfanew + 24 + 96 + 8*optHeader->NumberOfRvaAndSizes + 40*i, 
sectHeader[i]);
                  if(b == 1) {
                        int numPages = sectHeader[i]-
>SizeOfRawData/0x1000;
                        printf("\nThere are %d pages in this 
section\n\n", numPages);
                        int j;
                        for(j = 0;j < numPages;j++){
                              int vAddr = (long) peb->ImageBaseAddress 
+ (long) sectHeader[i]->VirtualAddress + 0x1000*j;
                              int pageAddr = virtual_to_physical((long) 
vAddr,eprocess->Pcb.DirectoryTableBase[0], in, size);
                              if(pageAddr != -1){
                                    printf("0x%.8x (0x%.8x)\n",  vAddr, 
pageAddr);
                              }
                        }
                  }
            }
      }
}

/**********************************************************************
```

```
Prints a report of information found in the Eprocess block.
Input: Eprocess block, count, offset
Output: Process information
**********************************************************************/
void print_process_report(struct EPROCESS *eprocess, int count, int
offset) {
      char created[32];
      win_time(eprocess->CreateTime, created);
      printf("%3d |%-20s|%8d|%8d|%25s| %10.8p | %10.8p |\n",
            count,
            eprocess->ImageFileName,
            eprocess->UniqueProcessId,
            eprocess->InheritedFromUniqueProcessId,
            created,
            offset,
            eprocess->Pcb.DirectoryTableBase[0]);
      fflush(NULL);
}

/**********************************************************************
Calculates the size of the Image file.  Used to determin the VAT
method.
Input: Image file
Output: Size of file
**********************************************************************/
long file_size(FILE *f){
      long cur_pos, length;
      cur_pos = ftell(f);
      fseek(f, 0, SEEK_END);        // set pointer to end of file
      length=ftell(f);        // offset in bytes from file's beginning
      fseek(f, cur_pos, SEEK_SET);  // restore original position
      return length;
}

/**********************************************************************
Checks if memory block contains a valid process.
Input: Eprocess block
Output: 1 if not process, 0 if it is
**********************************************************************/
int check_process(struct EPROCESS *procTest){
      int check = 1;
      if((procTest->Pcb.Header.Type != 0x03) || (procTest-
>Pcb.Header.Size != 0x1b)){
                  check = 0;
            } else if (procTest->Pcb.DirectoryTableBase[0] == 0){
                  check = 0;
                  } else if ((procTest->Pcb.DirectoryTableBase[0] %
0x1000) != 0){
                        check = 0;
                        } else if(((long) procTest-
>Pcb.ThreadListHead.Flink < KERNEL_OFFSET) || ((long) procTest-
>Pcb.ThreadListHead.Blink < KERNEL_OFFSET)){
                              check = 0;
                              } else if((procTest-
>WorkingSetLock.Event.Type != 0x01) || (procTest-
>WorkingSetLock.Event.Size != 0x04)){
                                    check = 0;
                                    }
      return check;
}

/***********************************************************************
Scans memory image for the active process head.  Used when enumerating
process list.
Input: File and offset to start search
Output: Offset to active process head
***********************************************************************/
int get_active_process_head(FILE * in, int offset){
      int check = 1;
      struct EPROCESS *procTest;
      procTest = malloc(sizeof(struct EPROCESS));
      while(!feof(in)){
            check = 1;
            fseek(in,offset,SEEK_SET);
            fread(procTest,SIZEOF_PROC,1,in);
            if((check_process(procTest) == 1) && (strcmp(procTest-
>ImageFileName,"System") == 0)){
                  free(procTest);
                  return offset;
            }
            offset +=8;
      }
      free(procTest);
      return 0;
}

/***********************************************************************
Prints process image information to the screen.
Input: File, size, offset to process block as ascii hex, binary
pagefile present
Output: Prints process reports
***********************************************************************/
void dump_process_image(FILE * in,long size, char *process_address, int
b) {
      struct EPROCESS *procCounter;
      int count = 0;
      long procAddr = hstr_i(process_address); //0x21fe020 notepad.exe
laptopMem.dump
      procCounter = malloc(sizeof(struct EPROCESS));
      fseek(in,procAddr,SEEK_SET);
      fread(procCounter,sizeof(struct EPROCESS),1,in);
      if(check_process(procCounter) == 1){
            printf("    |%-20s|%8s|%8s|%-25s| %-10s | %-10s
|\n","Name","Pid","PPid","       Time","  Offset","    PDB");
            printf("     --------------------  -------  -------  ------
------------------ ------------ ------------  \n");
            print_process_report(procCounter, count, procAddr);
            print_image_info(procCounter, in, size, b);
            free(procCounter);
            printf("\n\n");
      }
      else {
            printf("Process not Found!");
            printf("\n\n");
      }
}

/**********************************************************************
Uses the location of the Kernel Executable Image to identify which OS
is running.
Input: Image Base Address
Output: Operating System
**********************************************************************/
void identify_kernel(long imgBaseOffset) {
      switch (imgBaseOffset) {
            case 0x400000:
                  printf("Windows 2000\n");
                  break;
            case 0x4d4000:
                  printf("Windows XP\n");
                  break;
            case 0x4d0000:
                  printf("Windows XP\n");
                  break;
            case 0x4d5000:
                  printf("Windows XP\n");
                  break;
            case 0xa02000:
                  printf("Windows XP\n");
                  break;
            case 0x4d7000:
                  printf("Windows XP Service Pack 2\n");
                  break;
            case 0x4de000:
                  printf("Windows 2003\n");
                  break;
            case 0x800000:
                  printf("Windows 2003 Service Pack 1\n");
                  break;
            case 0x2000000:
                  printf("Windows Vista Beta 2\n");
                  break;
            case 0x1800000:
                  printf("Windows Vista RC1\n");
                  break;
            }
}

/**********************************************************************
Looks for Kernel Address Space to identify the Operating System.
Input: FIle, Size, Image Base Address, binary verbose
Output: Operating System and Kernel Data
**********************************************************************/
void check_kernel(FILE * in,long size, long imgBaseOffset, int b) {
      struct _IMAGE_DOS_HEADER *dosHeader;
      dosHeader = malloc(sizeof(struct _IMAGE_DOS_HEADER));
      fseek(in,(long) imgBaseOffset, SEEK_SET);
      fread(dosHeader,64,1,in);
      if(dosHeader->e_magic == 0x5a4d){
            identify_kernel(imgBaseOffset);
            if(b == 1) {
                  printf("\nImage Base Offset:
0x%x\n",imgBaseOffset);
                  print_dos_header(in,dosHeader,imgBaseOffset);
                  struct _IMAGE_FILE_HEADER *fileHeader;
                  fileHeader = malloc(sizeof(struct
_IMAGE_FILE_HEADER));
                  fseek(in,(long) imgBaseOffset + dosHeader->e_lfanew +
4, SEEK_SET);
                  fread(fileHeader,sizeof(struct
_IMAGE_FILE_HEADER),1,in);
                  print_file_header(fileHeader);
                  struct _IMAGE_OPTIONAL_HEADER *optHeader;
                  optHeader = malloc(sizeof(struct
_IMAGE_OPTIONAL_HEADER));
                  fseek(in,imgBaseOffset + dosHeader->e_lfanew +
24,SEEK_SET);
                  fread(optHeader,sizeof(struct
_IMAGE_OPTIONAL_HEADER),1,in);
                  print_optional_header(optHeader);
                  print_data_directories(optHeader);
                  printf("\n\t\tSection Information\n");
                  struct _IMAGE_SECTION_HEADER *sectHeader[fileHeader-
>NumberOfSections];
                  printf("\nName     Virt Size   Virt Addr   rData Ofs
rData Size  Char\n");
                  printf("----     ---------   ---------   ---------
----------  ----\n");
                  int i = 0;
                  for(i;i < fileHeader->NumberOfSections;i++){
                        sectHeader[i] = malloc(sizeof(struct
_IMAGE_SECTION_HEADER));
                        print_section_data(in, imgBaseOffset +
dosHeader->e_lfanew + 24 + 96 + 8*optHeader->NumberOfRvaAndSizes +
40*i, sectHeader[i]);
                  }
            }
      }
      free(dosHeader);
}

/*********************************************************************
Performs OS detection of a RAM dump by locating the kernel base address
and parsing the ResourceTable from the PE file located at that address.
Input: File, Size, binary verbose
Output: Operating System
*********************************************************************/
void osid(FILE * in,long size,int b){
      int os_type[10] =
{0x80400000,0x804d4000,0x804d0000,0x804d5000,0x80a02000,0x804d7000,0x80
4de000,0x80800000,0x82000000,0x81800000};
      int i = 0;
      int k_address = 0;
      for(i;i < 10;i++){
            k_address = os_type[i] - KERNEL_OFFSET;
            check_kernel(in, size, k_address, b);
      }
}
```

```
/**********************************************************************
Gets the process located at the process address from the raw memory
image.
Input: File, Size, Process Offset, empty Eprocess Block
Output: Eprocess block
**********************************************************************/
void get_process(FILE *in,long size, int procAddr, struct EPROCESS
*proc) {
      fseek(in,procAddr,SEEK_SET);
      fread(proc,sizeof(struct EPROCESS),1,in);

}

/**********************************************************************
Dumps the process memory from the Page Directory.
Input: File, Page File, Size, Process offset as char
Output: Process report and memory contents to file
**********************************************************************/
void dump_process_memory(FILE *in, FILE *pageFile, long size, char
*process_address) {
      long procAddr = hstr_i(process_address); //0x21fe020 notepad.exe
laptopMem.dump
      struct EPROCESS *proc;
      proc = malloc(sizeof(struct EPROCESS));
      get_process(in, size, procAddr, proc);
      if(check_process(proc) == 1){
            printf("    |%-20s|%8s|%8s|%-25s| %-10s | %-10s
|\n","Name","Pid","PPid","       Time","  Offset","    PDB");
            printf("     -------------------  -------  -------  ------
------------------ ----------- -----------  \n");
            print_process_report(proc, 0, procAddr);
            printf("\n");
            get_memory(proc, in, pageFile, size);
            free(proc);
            printf("\n\n");
      }
      else {
            printf("Process not Found!");
            printf("\n\n");
      }
}

/**********************************************************************
Gets the number of pages in memory.
Input: File, Page File, Size, Process offset as char
Output: Memory Stats
**********************************************************************/
void get_process_memory_stats(FILE *in,FILE *pageFile, long size, char
*process_address) {
      long procAddr = hstr_i(process_address);
      struct EPROCESS *proc;
      proc = malloc(sizeof(struct EPROCESS));
      get_process(in, size, procAddr, proc);
      if(check_process(proc) == 1){
            printf("    |%-20s|%8s|%8s|%-25s| %-10s | %-10s
|\n","Name","Pid","PPid","       Time","  Offset","    PDB");
```

```
            printf("     --------------------  -------  -------  ------
------------------ ------------ ------------  \n");
            print_process_report(proc, 0, procAddr);
            get_memory_stats(proc, in, pageFile, size);
            free(proc);
            printf("\n\n");
      }
      else {
            printf("Process not Found!");
            printf("\n\n");
      }
}

/*********************************************************************
Translates Virtual addresses to Physical Addresses.
Input: File, Size, Process offset as char
Output: Physical offset
*********************************************************************/
void translate_address(FILE *in, long size, char *pAddr, char *vAddr) {
      long procAddr = hstr_i(pAddr);
      long virtAddr = hstr_i(vAddr);
      struct EPROCESS *proc;
      if(virtAddr == 0){
            printf("Not a Valid Address\n");
            exit(0);
      }
      proc = malloc(sizeof(struct EPROCESS));
      get_process(in, size, procAddr, proc);
      if(check_process(proc) == 1){
            printf("    |%-20s|%8s|%8s|%-25s| %-10s | %-10s
|\n","Name","Pid","PPid","       Time","  Offset","    PDB");
            printf("     --------------------  -------  -------  ------
------------------ ------------ ------------  \n");
            print_process_report(proc, 0, procAddr);
            printf("\n");
            printf("Virtual Address:\t\t0x%.8x\n",virtAddr);
            printf("Physical
Address:\t\t0x%.8x\n",virtual_to_physical(virtAddr,proc-
>Pcb.DirectoryTableBase[0],in, size));
            free(proc);
            printf("\n\n");
      }
      else {
            printf("Process not Found!");
            printf("\n\n");
      }
}

/*********************************************************************
Given the pointer to the first process in the active process list, this
function follows the list of active processes.  Default value for the
pointer is 0.  If a valid process is not found it will continue to
search
the image from the address of the pointer.
Input: File, Size, Process offset
Output: List of processes
*********************************************************************/
```

```
void process_list(FILE * in,long size,long pAddr) {
      struct EPROCESS *proc;
      int count = 0;
      long procAddr = 0;
      procAddr = get_active_process_head(in, pAddr);
      proc = malloc(sizeof(struct EPROCESS));
      get_process(in, size, procAddr, proc);
      if((check_process(proc) == 1) && (strcmp(proc-
>ImageFileName,"System") == 0)) {
            printf("    |%-20s|%8s|%8s|%-25s| %-10s | %-10s
|\n","Name","Pid","PPid","       Time","  Offset","    PDB");
            printf("     --------------------  -------  -------  ------
------------------ ----------- -----------  \n");
            print_process_report(proc, count, procAddr);
            while(1) {
                  count++;
                  procAddr = virtual_to_physical((long) proc-
>ActiveProcessLinks.Flink,proc->Pcb.DirectoryTableBase[0],in, size)-
0x88;
                  get_process(in, size, procAddr, proc);
                  if(check_process(proc) != 1)
                        break;
                  print_process_report(proc, count, procAddr);
            }
      } else {
            printf("System process not found");
            exit(0);
      }
      free(proc);
      printf("\n\n");
}


/*********************************************************************
Identifies the name and size of the memory image.
Input: File, path to image, size
Output: Name and Size
*********************************************************************/
void identify_image(FILE * in, char *name,long size){
      char * format =
            "\nImage name: %21s\n"
            "Image Size: %12d Bytes\n\n";
      printf(format, name, size);
}

/*********************************************************************
Scans the image for process objects.
Input: file and size
Output: process scan
*********************************************************************/
void process_scan(FILE * in,long size) {
      int count = 0;
      int offset = 0;
      struct EPROCESS *procTest;
      procTest = malloc(sizeof(struct EPROCESS));
      printf("    |%-20s|%8s|%8s|%-25s| %-10s | %-10s
|\n","Name","Pid","PPid","       Time","  Offset","    PDB");
```

```
      printf("     --------------------  -------  -------  -----------
------------ ------------ -----------  \n");
      while(!feof(in)){
            get_process(in, size, offset, procTest);
            if(check_process(procTest) == 1){
                  print_process_report(procTest, count, offset);
                  count++;
            }
            offset +=8;
      }
      free(procTest);
      printf("\n\n");
}

/***********************************************************************
Main Function.
***********************************************************************/
int main( int argc, char **argv ) {

      int c = 0;
      FILE * imgFile = NULL;
      FILE * pageFile = NULL;
      long  procAddr = 0;
      if(argc < 3){
            print_usage();
            exit(0);
      }
      imgFile = fopen(argv[2],"rb");
      if( imgFile == NULL ){
                  fprintf(stderr,"Error: Could not open
%s!\n",argv[2]);
            exit(0);
      }
      int fileSize = file_size(imgFile);
      identify_image(imgFile,argv[2],fileSize);
      while ((c = getopt (argc, argv, "lstdipm")) != -1)
            switch (c) {
                  case 'l':
                        if(argc > 3){
                              procAddr = hstr_i(argv[3]);
                        }
                        process_list(imgFile, fileSize, procAddr);
                        break;
                  case 's':
                        process_scan(imgFile, fileSize);
                              break;
                  case 't':
                        if(argc > 4){
                              translate_address(imgFile, fileSize,
argv[3],argv[4]);
                        } else print_usage();
                              break;
                  case 'd':
                        if((argc > 4) && (strcmp(argv[4],"-pages") ==
0)) {
                              dump_process_image(imgFile, fileSize,
argv[3], 1);
```

```
                        } else dump_process_image(imgFile, fileSize,
argv[3], 0);
                              break;
                  case 'i':
                        if((argc > 3) && (strcmp(argv[3],"-verbose") ==
0)) {
                              osid(imgFile, fileSize, 1);
                        } else osid(imgFile, fileSize, 0);
                              break;
                  case 'p':
                        get_process_memory_stats(imgFile, pageFile,
fileSize, argv[3]);
                              break;
                  case 'm':
                        if(argc > 4){
                              pageFile = fopen(argv[4],"rb");
                        }
                        dump_process_memory(imgFile, pageFile,
fileSize, argv[3]);
                              break;
                  default:
                        print_usage();
                        break;
            }
      fclose(imgFile);
      return 0;
}
```

APPENDIX C. TEST RESULTS

A. TEST RESULTS

Eproc.exe has been tested against WPMOA [7] and Volatility [4]. The table below shows the results of the tests.

Test		Eproc	WPMOA [7]	Volatility [4]
List		38 procs	38 procs	38 procs
Scan		78 procs	N/A	75 procs
Mem dump	Notepad.exe	87 MB	20 MB	N/A

These tests show that processes that have been exited or removed from the active process list by some other means can be found by scanning the image for process block structures and that it gives a better picture of what was running on the system at the time the image was taken. This is a more forensically sound method of finding the processes in a memory image. The next test is the memory retrieved using Eproc over WPMOA [7]. The later used the working set list to find memory pages used by a process and only found 20 MB of memory for notepad.exe. Eproc on the other hand dumps all non-global pages from the page directory and when combined with the more robust method of address translation mentioned above was able to locate 87 MB of data. Improvements over WPMOA are an algorithm built in that will locate the Active Process List head, a scanning function used to locate processes not it the list, processing of PE header information, the ability to locate data in the swap file if available.

B. SAMPLE OUTPUT

1. Process List

```
Image name: /home/jstimson/Images/laptopMem.dump
Image Size:    536776704 Bytes

    |Name                |     Pid|    PPid|       Time              |   Offset   |     PDB    |
     --------------------  -------  -------  ------------------------ ------------ ------------
  0 |System              |       4|       0|  2009-04-22 19:24:48    | 0x023ca830 | 0x00039000 |
  1 |smss.exe            |    1628|       4|  2007-08-24 22:10:52    | 0x021bb8f8 | 0x18a12000 |
  2 |csrss.exe           |    1700|    1628|  2007-08-24 22:10:54    | 0x020adc08 | 0x1ac4b000 |
  3 |winlogon.exe        |    1724|    1628|  2007-08-24 22:11:01    | 0x0228b2c8 | 0x1efb0000 |
  4 |services.exe        |    1768|    1724|  2007-08-24 22:11:02    | 0x022851f8 | 0x1f0fc000 |
  5 |lsass.exe           |    1780|    1724|  2007-08-24 22:11:02    | 0x0213dda0 | 0x1f0b6000 |
  6 |svchost.exe         |    1948|    1768|  2007-08-24 22:11:05    | 0x02260c40 | 0x1f7a8000 |
  7 |svchost.exe         |    1996|    1768|  2007-08-24 22:11:06    | 0x02124da0 | 0x1f8b2000 |
  8 |svchost.exe         |     492|    1768|  2007-08-24 22:11:06    | 0x021a1020 | 0x1fb38000 |
  9 |svchost.exe         |     612|    1768|  2007-08-24 22:11:06    | 0x021355f8 | 0x1fce0000 |
 10 |svchost.exe         |     840|    1768|  2007-08-24 22:11:07    | 0x021bfda0 | 0x0073a000 |
 11 |ccSetMgr.exe        |    1120|    1768|  2007-08-24 22:11:08    | 0x02123358 | 0x0394f000 |
 12 |explorer.exe        |    1228|    1168|  2007-08-24 22:11:08    | 0x02280460 | 0x08dc3000 |
 13 |ccEvtMgr.exe        |    1248|    1768|  2007-08-24 22:11:08    | 0x01fae638 | 0x03f80000 |
 14 |spoolsv.exe         |    1448|    1768|  2007-08-24 22:11:09    | 0x020ecbe0 | 0x0b648000 |
 15 |DefWatch.exe        |    1680|    1768|  2007-08-24 22:11:18    | 0x01ee4518 | 0x0c27f000 |
 16 |GoogleUpdaterSe     |    1848|    1768|  2007-08-24 22:11:18    | 0x01fb9478 | 0x0c224000 |
 17 |mdm.exe             |    1912|    1768|  2007-08-24 22:11:18    | 0x0226ac78 | 0x0c209000 |
 18 |svchost.exe         |     284|    1768|  2007-08-24 22:11:18    | 0x02105be0 | 0x0d6f1000 |
 19 |Rtvscan.exe         |     692|    1768|  2007-08-24 22:11:24    | 0x01f08be0 | 0x0d8f5000 |
 20 |vmware-authd.ex     |     728|    1768|  2007-08-24 22:11:25    | 0x01eecda0 | 0x0da7b000 |
 21 |vmount2.exe         |     796|    1768|  2007-08-24 22:11:25    | 0x01f1d258 | 0x0dc76000 |
 22 |vmnat.exe           |     772|    1768|  2007-08-24 22:11:25    | 0x01ec9c10 | 0x0deae000 |
 23 |vmnetdhcp.exe       |    1824|    1768|  2007-08-24 22:11:25    | 0x01e20a28 | 0x0df63000 |
 24 |vmserverdWin32.     |     356|    1768|  2007-08-24 22:12:25    | 0x01e076a0 | 0x0f366000 |
 25 |alg.exe             |    3088|    1768|  2007-08-24 22:12:33    | 0x01df35e8 | 0x1186c000 |
 26 |ccApp.exe           |    3580|    1228|  2007-08-24 22:12:35    | 0x01e34020 | 0x12025000 |
 27 |VPTray.exe          |    3812|    1228|  2007-08-24 22:12:36    | 0x021e1738 | 0x1447d000 |
 28 |acrotray.exe        |    3820|    1228|  2007-08-24 22:12:36    | 0x01f11da0 | 0x14e91000 |
 29 |jusched.exe         |    3924|    1228|  2007-08-24 22:12:40    | 0x01ef3438 | 0x131e6000 |
 30 |ctfmon.exe          |    1136|    1228|  2007-08-24 22:12:42    | 0x020dd558 | 0x1411e000 |
 31 |GoogleUpdater.e     |    2300|    1228|  2007-08-24 22:12:44    | 0x01de9a68 | 0x14e8b000 |
 32 |FNPLicensingSer     |    2728|    1768|  2007-08-24 22:12:50    | 0x01eef530 | 0x1c4c8000 |
 33 |cmd.exe             |     984|    1228|  2007-08-24 22:13:16    | 0x01fad820 | 0x03d23000 |
 34 |bash.exe            |    1092|     984|  2007-08-24 22:13:17    | 0x01e53020 | 0x1577a000 |
 35 |firefox.exe         |    2440|    1228|  2007-08-24 23:24:30    | 0x01dd3a60 | 0x0a62d000 |
 36 |notepad.exe         |     720|    1228|  2007-08-24 23:24:46    | 0x021fe020 | 0x0cb45000 |
 37 |cmd.exe             |    2892|    1228|  2007-08-24 23:25:11    | 0x01da9248 | 0x1e3e0000 |
 38 |dd.exe              |    2232|    2892|  2007-08-24 23:26:55    | 0x01d7d020 | 0x0c4aa000 |
```

2. Process Scan

```
Image name: /home/jstimson/Images/laptopMem.dump
Image Size:    536776704 Bytes

    |Name                |     Pid|    PPid|       Time             |  Offset    |    PDB     |
     --------------------  -------  -------  ------------------------ ------------ ------------
  0 |Idle                |       0|       0|  2009-04-22 19:24:48    | 0x00559080 | 0x00039000 |
  1 |dd.exe              |    2232|    2892|  2007-08-24 23:26:55    | 0x01d7d020 | 0x0c4aa000 |
  2 |cmd.exe             |    2492|    1228|  2007-08-24 22:24:48    | 0x01d846b0 | 0x07a74000 |
  3 |cmd.exe             |    2892|    1228|  2007-08-24 23:25:11    | 0x01da9248 | 0x1e3e0000 |
  4 |bash.exe            |    2404|    1092|  2007-08-24 23:18:09    | 0x01dc8218 | 0x06c32000 |
  5 |firefox.exe         |    2440|    1228|  2007-08-24 23:24:30    | 0x01dd3a60 | 0x0a62d000 |
  6 |GoogleUpdater.e     |    2300|    1228|  2007-08-24 22:12:44    | 0x01de9a68 | 0x14e8b000 |
  7 |alg.exe             |    3088|    1768|  2007-08-24 22:12:33    | 0x01df35e8 | 0x1186c000 |
  8 |vmserverdWin32.     |     356|    1768|  2007-08-24 22:12:25    | 0x01e076a0 | 0x0f366000 |
  9 |vmnetdhcp.exe       |    1824|    1768|  2007-08-24 22:11:25    | 0x01e20a28 | 0x0df63000 |
 10 |ccApp.exe           |    3580|    1228|  2007-08-24 22:12:35    | 0x01e34020 | 0x12025000 |
 11 |bash.exe            |    1092|     984|  2007-08-24 22:13:17    | 0x01e53020 | 0x1577a000 |
 12 |vmnat.exe           |     772|    1768|  2007-08-24 22:11:25    | 0x01ec9c10 | 0x0deae000 |
 13 |DefWatch.exe        |    1680|    1768|  2007-08-24 22:11:18    | 0x01ee4518 | 0x0c27f000 |
 14 |vmware-authd.ex     |     728|    1768|  2007-08-24 22:11:25    | 0x01eecda0 | 0x0da7b000 |
 15 |FNPLicensingSer     |    2728|    1768|  2007-08-24 22:12:50    | 0x01eef530 | 0x1c4c8000 |
 16 |jusched.exe         |    3924|    1228|  2007-08-24 22:12:40    | 0x01ef3438 | 0x131e6000 |
 17 |Rtvscan.exe         |     692|    1768|  2007-08-24 22:11:24    | 0x01f08be0 | 0x0d8f5000 |
 18 |acrotray.exe        |    3820|    1228|  2007-08-24 22:12:36    | 0x01f11da0 | 0x14e91000 |
 19 |vmount2.exe         |     796|    1768|  2007-08-24 22:11:25    | 0x01f1d258 | 0x0dc76000 |
 20 |cmd.exe             |     984|    1228|  2007-08-24 22:13:16    | 0x01fad820 | 0x03d23000 |
 21 |ccEvtMgr.exe        |    1248|    1768|  2007-08-24 22:11:08    | 0x01fae638 | 0x03f80000 |
 22 |GoogleUpdaterSe     |    1848|    1768|  2007-08-24 22:11:18    | 0x01fb9478 | 0x0c224000 |
 23 |csrss.exe           |    1700|    1628|  2007-08-24 22:10:54    | 0x020adc08 | 0x1ac4b000 |
 24 |ctfmon.exe          |    1136|    1228|  2007-08-24 22:12:42    | 0x020dd558 | 0x1411e000 |
 25 |spoolsv.exe         |    1448|    1768|  2007-08-24 22:11:09    | 0x020ecbe0 | 0x0b648000 |
 26 |svchost.exe         |     284|    1768|  2007-08-24 22:11:18    | 0x02105be0 | 0x0d6f1000 |
 27 |ccSetMgr.exe        |    1120|    1768|  2007-08-24 22:11:08    | 0x02123358 | 0x0394f000 |
 28 |svchost.exe         |    1996|    1768|  2007-08-24 22:11:06    | 0x02124da0 | 0x1f8b2000 |
 29 |svchost.exe         |     612|    1768|  2007-08-24 22:11:06    | 0x021355f8 | 0x1fce0000 |
 30 |lsass.exe           |    1780|    1724|  2007-08-24 22:11:02    | 0x0213dda0 | 0x1f0b6000 |
 31 |svchost.exe         |     492|    1768|  2007-08-24 22:11:06    | 0x021a1020 | 0x1fb38000 |
 32 |smss.exe            |    1628|       4|  2007-08-24 22:10:52    | 0x021bb8f8 | 0x18a12000 |
 33 |svchost.exe         |     840|    1768|  2007-08-24 22:11:07    | 0x021bfda0 | 0x0073a000 |
 34 |VPTray.exe          |    3812|    1228|  2007-08-24 22:12:36    | 0x021e1738 | 0x1447d000 |
 35 |dd.exe              |    3284|    2404|  2007-08-24 23:18:09    | 0x021f30e8 | 0x1f71f000 |
 36 |notepad.exe         |     720|    1228|  2007-08-24 23:24:46    | 0x021fe020 | 0x0cb45000 |
 37 |svchost.exe         |    1948|    1768|  2007-08-24 22:11:05    | 0x02260c40 | 0x1f7a8000 |
 38 |mdm.exe             |    1912|    1768|  2007-08-24 22:11:18    | 0x0226ac78 | 0x0c209000 |
 39 |explorer.exe        |    1228|    1168|  2007-08-24 22:11:08    | 0x02280460 | 0x08dc3000 |
 40 |services.exe        |    1768|    1724|  2007-08-24 22:11:02    | 0x022851f8 | 0x1f0fc000 |
 41 |winlogon.exe        |    1724|    1628|  2007-08-24 22:11:01    | 0x0228b2c8 | 0x1efb0000 |
 42 |Acrobat.exe         |    1332|    1228|  2007-08-24 22:22:37    | 0x022f27b8 | 0x0924c000 |
 43 |System              |       4|       0|  2009-04-22 19:24:48    | 0x023ca830 | 0x00039000 |
 44 |svchost.exe         |    1948|    1768|  2007-08-24 22:11:05    | 0x0257ec40 | 0x1f7a8000 |
 45 |jusched.exe         |    3924|    1228|  2007-08-24 22:12:40    | 0x03416438 | 0x131e6000 |
 46 |ccSetMgr.exe        |    1120|    1768|  2007-08-24 22:11:08    | 0x040bd358 | 0x0394f000 |
 47 |bash.exe            |    2404|    1092|  2007-08-24 23:18:09    | 0x04229218 | 0x06c32000 |
 48 |GoogleUpdaterSe     |    1848|    1768|  2007-08-24 22:11:18    | 0x045ad478 | 0x0c224000 |
 49 |notepad.exe         |     720|    1228|  2007-08-24 23:24:46    | 0x046db020 | 0x0cb45000 |
 50 |vmserverdWin32.     |     356|    1768|  2007-08-24 22:12:25    | 0x059886a0 | 0x0f366000 |
 51 |bash.exe            |    2404|    1092|  2007-08-24 23:18:09    | 0x06043218 | 0x06c32000 |
 52 |mdm.exe             |    1912|    1768|  2007-08-24 22:11:18    | 0x07488c78 | 0x0c209000 |
 53 |ccEvtMgr.exe        |    1248|    1768|  2007-08-24 22:11:08    | 0x08052638 | 0x03f80000 |
 54 |Acrobat.exe         |    1332|    1228|  2007-08-24 22:22:37    | 0x083017b8 | 0x0924c000 |
 55 |                    |       4|       0|  2009-04-22 19:24:48    | 0x09341998 | 0x00039000 |
 56 |Acrobat.exe         |    1332|    1228|  2007-08-24 22:22:37    | 0x0a7ed7b8 | 0x0924c000 |
 57 |ccSetMgr.exe        |    1120|    1768|  2007-08-24 22:11:08    | 0x0a8a3358 | 0x0394f000 |
 58 |ccEvtMgr.exe        |    1248|    1768|  2007-08-24 22:11:08    | 0x0a908638 | 0x03f80000 |
 59 |                    |       4|       0|  2009-04-22 19:24:48    | 0x0ae10998 | 0x00039000 |
 60 |explorer.exe        |    1228|    1168|  2007-08-24 22:11:08    | 0x0ae8e460 | 0x08dc3000 |
 61 |FNPLicensingSer     |    2728|    1768|  2007-08-24 22:12:50    | 0x0b3b2530 | 0x1c4c8000 |
 62 |VPTray.exe          |    3812|    1228|  2007-08-24 22:12:36    | 0x0b47e738 | 0x1447d000 |
 63 |ccApp.exe           |    3580|    1228|  2007-08-24 22:12:35    | 0x0c096020 | 0x12025000 |
 64 |bash.exe            |    1092|     984|  2007-08-24 22:13:17    | 0x0ccc5020 | 0x1577a000 |
 65 |                    |       4|       0|  2009-04-22 19:24:48    | 0x0cd1f998 | 0x00039000 |
 66 |spoolsv.exe         |    1448|    1768|  2007-08-24 22:11:09    | 0x0d055be0 | 0x0b648000 |
 67 |svchost.exe         |     492|    1768|  2007-08-24 22:11:06    | 0x0e8bc020 | 0x1fb38000 |
 68 |vmware-authd.ex     |     728|    1768|  2007-08-24 22:11:25    | 0x0ea4fda0 | 0x0da7b000 |
 69 |csrss.exe           |    1700|    1628|  2007-08-24 22:10:54    | 0x0ed23c08 | 0x1ac4b000 |
 70 |vmnat.exe           |     772|    1768|  2007-08-24 22:11:25    | 0x0ee9cc10 | 0x0deae000 |
 71 |svchost.exe         |     284|    1768|  2007-08-24 22:11:18    | 0x0f08fbe0 | 0x0d6f1000 |
 72 |bash.exe            |    2404|    1092|  2007-08-24 23:18:09    | 0x0f6a5218 | 0x06c32000 |
 73 |vmount2.exe         |     796|    1768|  2007-08-24 22:11:25    | 0x0fa90258 | 0x0dc76000 |
 74 |winlogon.exe        |    1724|    1628|  2007-08-24 22:11:01    | 0x0fc692c8 | 0x1efb0000 |
 75 |svchost.exe         |     612|    1768|  2007-08-24 22:11:06    | 0x0fccf5f8 | 0x1fce0000 |
 76 |GoogleUpdaterSe     |    1848|    1768|  2007-08-24 22:11:18    | 0x0feee478 | 0x0c224000 |
 77 |DefWatch.exe        |    1680|    1768|  2007-08-24 22:11:18    | 0x0ff97518 | 0x0c27f000 |
 78 |notepad.exe         |     720|    1228|  2007-08-24 23:24:46    | 0x10c9e020 | 0x0cb45000 |
```

3. Process Information

```
Image name: /home/jstimson/Images/laptopMem.dump
Image Size:    536776704 Bytes

    |Name                |     Pid|    PPid|       Time              |   Offset   |     PDB    |
     --------------------  -------  -------  ------------------------ ------------ ------------
  0 |bash.exe            |    1092|     984|  2007-08-24 22:13:17    | 0x01e53020 | 0x1577a000 |

Imgage Base Offset:             0x1b393000

               DOS Header
e_magic:                        0x5a4d
e_lfanew:                       0x80
SIGNATURE:                      0x4550

               File Header
Machine Type:                   IMAGE_FILE_MACHINE_I386
Number of Sections:             5
Time Stamp:                     2009-04-22 19:26:45
Size of Optional Header:        0xe0 (224 bytes)
Characteristics:
        IMAGE_FILE_RELOCS_STRIPPED
        IMAGE_FILE_EXECUTABLE_IMAGE
        IMAGE_FILE_LINE_NUMS_STRIPPED
        IMAGE_FILE_LOCAL_SYMS_STRIPPED
        IMAGE_FILE_32BIT_MACHINE
        IMAGE_FILE_DEBUG_STRIPPED


               Optional Header
Magic Number:                   0x10b
Subsystem:                      IMAGE_SUBSYSTEM_WINDOWS_CUI
Image Base Address:             0x400000
Address of Entry Point:         0x1000 (RVA)
Code Base Address:              0x1000 (RVA)
Data Base Address:              0x5b000 (RVA)
Major OS Version:               4
Minor OS Version:               0
Size of Stack Reserved:         0x200000 (2097152 bytes)
Number of RVA and Sizes:        0x10

               Data Directory Information

Data Directory                  RVA            Size
--------------                  ---            ----
Import Directory                0x00076000     0x00003e3c

               Section Information

Name     Virt Size   Virt Addr   rData Ofs   rData Size  Char
----     ---------   ---------   ---------   ----------  ----
.text    0x00059200  0x00001000  0x00059400  0x00000400  0xe0000020
.data    0x00002400  0x0005b000  0x00002400  0x00059800  0xc0000040
.rdata   0x00013000  0x0005e000  0x00013200  0x0005bc00  0x40000040
.bss     0x00003200  0x00072000  0x00000000  0x00000000  0xc0000080
.idata   0x00003e00  0x00076000  0x00004000  0x0006ee00  0xc0000040
```

4. Operating System Identification -Verbose

```
        Image name: /home/jstimson/Images/laptopMem.dump
        Image Size:    536776704 Bytes

        Windows XP Service Pack 2

        Image Base Offset:             0x4d7000

                       DOS Header
        e_magic:                        0x5a4d
        e_lfanew:                       0xe8
        SIGNATURE:                      0x4550
                       File Header
        Machine Type:                   IMAGE_FILE_MACHINE_I386
        Number of Sections:             21
        Time Stamp:                     2009-04-22 19:26:45
        Size of Optional Header:        0xe0 (224 bytes)
        Characteristics:

               IMAGE_FILE_EXECUTABLE_IMAGE
               IMAGE_FILE_LINE_NUMS_STRIPPED
               IMAGE_FILE_LOCAL_SYMS_STRIPPED
               IMAGE_FILE_32BIT_MACHINE

                       Optional Header
        Magic Number:                   0x10b
        Subsystem:                      IMAGE_SUBSYSTEM_NATIVE
        Image Base Address:             0x400000
        Address of Entry Point:         0x1d55ce (RVA)
        Code Base Address:              0x580 (RVA)
        Data Base Address:              0x6a400 (RVA)
```

```
       Major OS Version:               5
       Minor OS Version:               1
       Size of Stack Reserved:         0x40000 (262144 bytes)
       Number of RVA and Sizes:        0x10

                      Data Directory Information

       Data Directory                  RVA            Size
       --------------                  ---            ----
       Export Directory                0x001aa380     0x0000b55d
       Import Directory                0x001f3a24     0x00000050
       Resource Directory              0x001f4380     0x00010708
       Base Relocation Table           0x00204b00     0x0000f984
       Debug Directory                 0x00071f00     0x00000038
       Load Configuration Directory    0x000535a0     0x00000040

                      Section Information

       Name     Virt Size   Virt Addr   rData Ofs   rData Size  Char
       ----     ---------   ---------   ---------   ----------  ----
       .text    0x00071900  0x00000580  0x00071a00  0x00000580  0x68000020
       POOLMI   0x00001200  0x00071f80  0x00001300  0x00071f80  0x68000020
       MISYSPTE 0x00000700  0x00073280  0x00000700  0x00073280  0x68000020
       POOLCODE 0x00001500  0x00073980  0x00001600  0x00073980  0x68000020
       .data    0x00016c00  0x00074f80  0x00016d00  0x00074f80  0xc8000040
       PAGE     0x000f8e00  0x0008bc80  0x000f8e80  0x0008bc80  0x60000020
       PAGEVRFY 0x0000f100  0x00192e80  0x0000f200  0x00192e80  0x60000020
       PAGEWMI  0x00001700  0x001a2080  0x00001800  0x001a2080  0x60000020
       PAGEKD   0x00004000  0x001a3880  0x00004080  0x001a3880  0x60000020
       .edata   0x0000b500  0x001aa380  0x0000b580  0x001aa380  0x40000040
       PAGEDATA 0x00001500  0x001b5900  0x00001580  0x001b5900  0xc0000040
       PAGEKD   0x0000c000  0x001b6e80  0x0000c080  0x001b6e80  0xc0000040
       PAGECONS 0x00000100  0x001c2f00  0x00000200  0x001c2f00  0xc0000040
       PAGEVRFC 0x00003400  0x001c3100  0x00003480  0x001c3100  0x40000040
       PAGEVRFD 0x00000600  0x001c6580  0x00000680  0x001c6580  0xc0000040
       INIT     0x0002d700  0x001c6c00  0x0002d780  0x001c6c00  0xe2000020
       .rsrc    0x00010700  0x001f4380  0x00010780  0x001f4380  0x40000040
       .reloc   0x0000f900  0x00204b00  0x0000fa00  0x00204b00  0x42000040
```

5. Memory Statistics

```
Image name: /home/jstimson/Images/laptopMem.dump
Image Size:    536776704 Bytes

    |Name                |     Pid|    PPid|       Time              |   Offset   |     PDB    |
     --------------------  -------  -------  ------------------------ ------------ ------------
  0 |dd.exe              |    2232|    2892|  2007-08-24 23:26:55    | 0x01d7d020 | 0x0c4aa000 |

Valid          2020
Prototype      50966
Global         32156
Paged          8400
Unknown        324
```

6. Address Translation

```
Image name: /home/jstimson/Images/laptopMem.dump
Image Size:    536776704 Bytes

    |Name                |     Pid|    PPid|       Time              |   Offset   |     PDB    |
     --------------------  -------  -------  ------------------------ ------------ ------------
  0 |dd.exe              |    2232|    2892|  2007-08-24 23:26:55    | 0x01d7d020 | 0x0c4aa000 |

Virtual Address:              0x00407000
Physical Address:             0x04a1a000
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] Chris Prosise; Kevin Mandia; Matt Pepe. *Incident Response and Computer Forensics, Second Edition*. McGraw-Hill Osborne Media, 2003.

[2] Brian D. Carrier; Joe Grand. “A Hardware-Based Memory Acquisition Procedure for Digital Investigations.” *Digital Investigation Journal*, Issue 1(1), February 2004.

[3] George M. Garner. “Forensic Acquisition Tools.” Internet: users.erols.com/gmgarner/forensics/, August 17, 2004.

[4] Nick Petroni; AAron Walters, Volatile Systems, Volatility http://www.volatilesystems.com/VolatileWeb/volatility.gsp. 25 February 2007.

[5] Mark E. Russinovich; David A. Solomon. *Microsoft® Windows® Internals, Fourth Edition: Microsoft Windows Server™ 2003, Windows XP, and Windows 2000*. Redmond, Washington: Microsoft Press, 2005.

[6] J. Rutkowska. "Beyond the CPU: Defeating Hardware Based RAM Acquisition Tools (Part I: AMD case)," Black Hat DC, February 2007.

[7] John Schultz. "Offline Forensic Analysis of Microsoft Windows XP Physical Memory," Naval Postgraduate School Thesis, September 2006.

[8] J. Kornblum. "Using Every Part of the Buffalo in Windows Memory Analysis," Digital Investigation Journal, January 2007.

[9] J. Solomon; E. Huebner; D. Bem; M. Szezynska. "User data persistence in physical memory," Digital Investigation, Volume 4, Issue 2, June 2007, pp 68-72.

[10] Chris Betz. Memparser, 1.0 edition, July 2006. http://sourceforge.net/projects/memparser/ 15 January 2007.

[11] Mariusz Burdach. An introduction to the windows memory forensic, 2005. http://forensic.seccure.net/pdf/introduction to windows memory forensic.pdf. 3 December 2007.

[12] Nick Petroni; Aaron Walters; Timothy Fraser; William Arbaugh. FATKit: A framework for the extraction and analysis of digital forensic data from volatile system memory. Digital Investigation, 3(4):197–210, December 2006.

[13] Nicholas P. Maclean. Acquisition and analysis of Windows© memory. Master’s thesis, University of Strathclyde, 2006. http://www.cis.strath.ac.uk/_nimaclea/fi/reports/windows memory.pdf. 20 November 2006.

[14] Andreas Schuster. Searching for processes and threads in Microsoft Windows© memory dumps. Digital Investigation, 3(S):10–16, August 2006. http://dfrws.org/2006/proceedings/2-Schuster.pdf. 26 September 2006.

[15] Tim Vidas. Forensic analysis of volatile memory stores. NEbraskaCERT Conference, August 2006. http://www.certconf.org/presentations/2006/files/RB3.pdf. 26 September 2006

[16] Tim Vidas. "The Acquisition and Analysis of Random Access Memory," Journal of Digital Forensic Practice, Volume 1, Issue 4 December 2006, pp 315 - 323.

[17] Jamie Butler. "DKOM (Direct Kernel Object Manipulation)," BlackHat Windows Security 2004, www.blackhat.com/presentations/win-usa-04/bh-win-04-butler.pdf. 14 January 2007.

[18] *The GNU Netcat Project*, http://netcat.sourceforge.net/, June 2008.

[19] *KnTDD*, KnTTools, GMG Systems, Inc. http://gmgsystemsinc.com/knttools/, June 2008.

[20] *Enter Sandman* – Japan Pacsec 2007, http://www.msuiche.net/pres/PacSec07-slides-0.4.pdf, December 2007.

[21] Cygwin^{tm}, http://www.cygwin.com, June 2008.

[22] *Backtrack 3*, Remote-Exploit, http://www.remote-exploit.org/backtrack.html, June 2008.THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

2. Chris S. Eagle
   Naval Postgraduate School
   Monterey, California

3. George W. Dinolt
   Naval Postgraduate School
   Monterey, California